An Investigation into the Solution to, and Evaluation of, Kakuro Puzzles

Ryan P. Davies
Division of Mathematics and Statistics
University of Glamorgan

A submission presented in
partial fulfilment of the requirements
of the University of Glamorgan/Prifysgol Morgannwg
for the degree of Master of Philosophy

September 2009

Acknowledgements

Foremost, I would like to express my sincere gratitude to my director of studies, Dr. Paul Roach and my supervisor, Dr. Stephanie Perkins for their continuous support and encouragement for the duration of my MPhil study. Thank you both for your patience, knowledge, motivation, suggestions and enthusiasm. I could not have hoped for a better supervisory team. Thank you also to Sian Jones for many helpful and constructive conversations and ideas.

I am also grateful to the Division of Mathematics and Statistics for giving me the opportunity to lecture across a broad range of subjects, which helped me develop as a teacher. Finally, my best regards to the Faculty of Advanced Technology administrative staff, my G501 officemates and friends and colleagues with whom I have shared my two years in G-Block.

Abstract

Kakuro puzzles, also known as Cross-Sum puzzles, are similar in structure to standard Crossword puzzles. They consist of grids, containing overlapping continuous runs that are exclusively either horizontal or vertical, with "clues" to the completion of numerical "words". The numerical clues take the form of specified run-totals, and a puzzle is solved by placing a value from a given valid range (usually $1, \ldots, 9$) into each cell. A valid solution is reached when every run sums to its specified total, and no run contains duplicate values. While most puzzles have only a single solution, longer runs may be satisfied using many different arrangements of values, leading to the puzzle having a deceptively large search space. The associated, popular Sudoku puzzle has been linked with important real-world applications including conflict free wavelength routing and timetabling, and more recently, coding theory due to its potential usefulness in the construction of erasure correction codes. It is possible that Kakuro puzzles will have similar applications, particularly in the construction of codes, where run-totals may form a generalised type of error check. This thesis presents an investigation into the properties of Kakuro puzzles, and considers the potential usefulness of Kakuro to real-world applications. Specifically, this thesis determines bounds on the number of valid grid arrangements, a partial enumeration of Kakuro puzzles and compares methods of automating the solution of Kakuro puzzles that incorporate, where possible, puzzle domain information.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Description

Kakuro puzzles consist of an $n \times m$ grid containing black and white cells. The top row and the left column of black cells are not usually included when describing the dimensions of a Kakuro grid. All white cells are initially empty and are organised into overlapping continuous runs that are exclusively either horizontal or vertical. A run-total, usually given in a black "clue" cell, is associated with each and every run, and the puzzle is solved by entering values (typically from the range $1, \ldots, 9$ inclusive) into the white cells such that each run sums to the specified total and no digit is duplicated in any run. Assuming only numbers in the range $1, \ldots, 9$ are used, a run can be between one and nine cells in length with a corresponding run-total in the range $1, \ldots, 45$, although the majority of published puzzles contain runs that are at least two cells in length. Fig. 1.1 shows a sample Kakuro puzzle consisting of a 5×5 grid. Puzzles of this type have often been likened to a typical Crossword puzzle, due to the fact that there are "clues" that specify correct numerical "words" within a grid structure. Unlike Crossword puzzles, Kakuro puzzles more easily transcend language barriers due to their use of number sequences; a similar observation has been made for Sudoku puzzles [61].

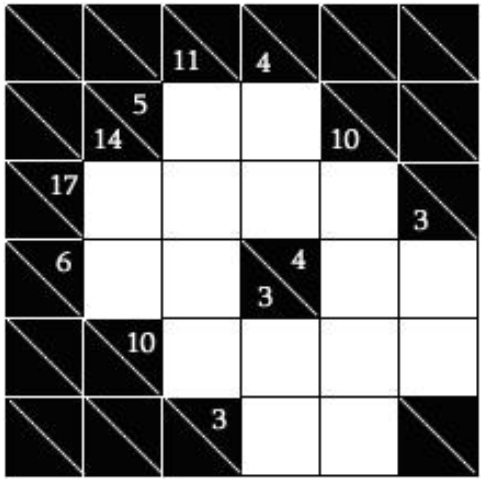

Figure 1.1: A typical Kakuro puzzle [23]

Most published puzzles are *well formed* [30], meaning that only one unique solution exists. Such puzzles are also called *promise-problems* (the promise being a unique solution) [7]. An example of a puzzle that possesses multiple solutions is shown in Fig. 1.2 below.

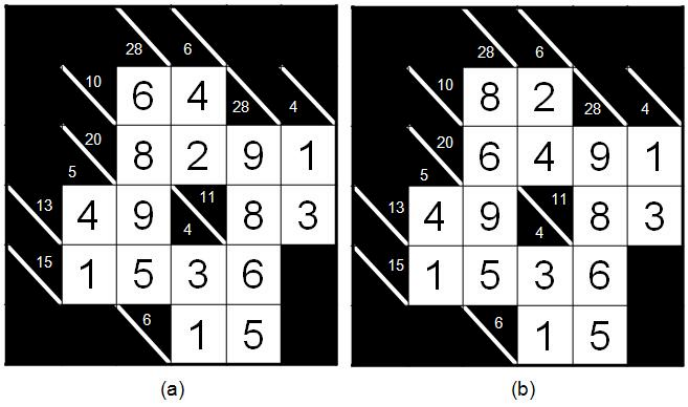

Figure 1.2: A 5×5 Kakuro Puzzle that possesses multiple solutions

Many puzzle grids have reflective or rotational symmetry, although this is only to improve the visual appearance of the grid [26]. Logical deduction and reasoning are employed by human solvers to solve such puzzles.

The name "Kakuro" comes from *"Kasan Karuso"*; the Japanese pronunciation of the En-

glish word "cross" appended to the Japanese word for "addition". This name was a part of a rebranding by Japan's Nikoli Puzzles Group of Dell Magazines' "Cross Sum" puzzles, as they were then known. Dell Magazines published such puzzles as early as 1966 [22]. However, Kakuro's huge popularity is recent; Kakuro puzzles first appeared in the United Kingdom on September 14th 2005 in The Guardian newspaper. Today, the popularity of Kakuro in Japan is reported second only to Sudoku [22].

Related puzzles include *"Cryptic Kakuro"* [69], in which alphametric clues must be solved as a prerequisite to the Kakuro puzzle itself; *"Cross-sum Sudoku"* [69], which combines the rules of standard Kakuro puzzles with the constraints of standard Sudoku puzzles; *"Cross Products"*, in which the black, "clue" cells suggest the product of values placed in a run, rather than their sum; *"Crosswords"*, where quick or cryptic clues guide the player to place words into each and every word-slot within a puzzle grid such that all clues are satisfied and all words interlock correctly and *"Survo Puzzles"* [45], in which all values in the range $1, \ldots, nm$ must be placed into an $n \times m$ grid, often containing givens, so as to satisfy all row and column sums which are provided above each column and to the left of each row. (Crossword and Survo puzzles will be examined in greater detail in Section 2.2.)

At present, very little has been published specifically on Kakuro and its related puzzles, however, their solution has been shown to be NP-Complete [56], through demonstrating the relationship between the Hamiltonian Path Problem, 3SAT (the restriction of the Boolean satisfiability problem) and Kakuro.

1.2 Aims

Recreational mathematical puzzles have previously been shown to link to real-world problems. An understanding of the usefulness of these puzzles has arisen from detailed analysis of their underlying properties. Latin Squares and Sudoku in particular have been suggested as applicable in several areas, including conflict free wavelength routing [16], timetabling design [24, 42] and experimental design [24]. Recently, it has been proposed that Sudoku puzzles have applications in coding theory due to their potential usefulness in the construction of erasure correction codes [62]. It may be the case that Kakuro-type puzzles may have similar applications to Sudoku, particularly in the construction of codes, where run-totals

may form a type of error check.

To ascertain any potential applicability of Kakuro, it is first necessary to establish its underlying mathematical properties. Both Kakuro and Sudoku contain some form of non-duplication constraints. Kakuro puzzles additionally possess run-total constraints, meaning placed values not only have to be distinct from others within a run, but must sum to the correct target. Additionally, Kakuro puzzles do not contain given values and cannot be linked to Latin squares; values do not necessarily appear a given number of times within a puzzle grid and additionally, may appear more than once in a row or column if and only if the repeated values in question appear in distinct runs within the row or column.

At present, little has been published on the properties of Kakuro puzzles, so this thesis will address the properties and puzzle-domain information of Kakuro puzzles. Such properties, together with constraint information will be used to enumerate the smallest puzzle grids, and the possible extension to larger grids will be examined. Hence it will be determined whether puzzle complexity will rise disproportionately to any increase in grid dimension. White cells within Kakuro puzzles can accept a varying number of values, depending on the runs to which it belongs.

While a well-formed puzzle has only one solution, the complexity of puzzles arises from the many ways in which runs and groups of runs may be completed. The puzzle constraints restrict the numbers of valid run and grid arrangements. This thesis examines in detail the effect of each constraint. Bounds are presented for the number of valid grid arrangements that exist for a given grid by using information about the number of values that can be placed in each cell. This will indicate whether a cell can be filled using a broad or narrow set of values. A generating function will be presented showing the total number of valid, unordered partitions of run-totals into a given number of cells within a run. Information concerning run lengths and totals will be used to construct candidate sets of values that may be legitimately assigned to cells.

It is likely that any real-world application of Kakuro will require an efficient means of automating the solution of Kakuro puzzles. This thesis also aims to establish the most appropriate approach for the automated solution of Kakuro puzzles, and to consider how well approaches might be extended to large puzzle grid sizes. Puzzle information will be used to

inform the development of algorithms for several standard search techniques. Following the application of a range of such automated search-based methods, basic inferences concerning puzzle properties are drawn.

1.3 Thesis Structure

In Chapter 2, terminology that will be used to describe a Kakuro puzzle grid and the puzzle properties is introduced. A review of the literature relating to Kakuro puzzles is then given. In the academic literature, very little work currently appears specifically on Kakuro puzzles apart from work published by the current author [13, 14, 15]. Therefore, emphasis will be placed on work published on similar and related puzzles and their potential applicability to Kakuro puzzles. Brief introductions to the rules of such related puzzles are also provided.

Chapter 3 initially explores the difficulty gradings assigned to puzzles, investigating whether they might be derived from the selection of runs alone (their lengths and the number of options that exist for their completion), or by some complex methods, such as the use of computer-based grid production algorithms. Bounds on the number of valid arrangements of values that can be placed within given grids are then considered. This chapter also ascertains which type of constraint has most effect in terms of improving such bounds. This analysis examines underlying puzzle properties that, where possible, will be later used in Chapter 5 to inform automated approaches to puzzle solution.

Chapter 4 describes standard methods for automating the solutions to problems. Each general approach is described and consideration is then given to how the approach may be implemented for Kakuro, hence enabling the evaluation of the usefulness of each approach. Section 4.1 describes implementations of exhaustive search (with and without backtracking), Section 4.2 outlines a local search approach and Section 4.3 introduces the use of metaheuristics, including tabu search and genetic algorithms. Section 4.4 describes a constraint based approach to problems, and a binary integer programming approach is implemented. The evaluation of these techniques suggests a depth-first exhaustive search approach with backtracking to be the most suitable for the automated solution of Kakuro puzzles.

Having identified in Chapter 4 the advantages of using a depth-first approach with back-

tracking to the automated solution of Kakuro puzzles, Chapter 5 outlines such an approach for the automated solution of Kakuro puzzles. In this chapter, two backtracking algorithms are described – an implementation through the use of a stack (Section 5.1) and a recursive implementation (Section 5.4). Pruning rules and heuristics, incorporated into both algorithms to improve the solution time, are described and implemented. These heuristics are used to direct the presented algorithms through search spaces via the cheapest path to a goal state. The heuristics exploit the features of the problem domain in order to reduce time spent examining a search space [53]. Effective pruning conditions are determined to reduce the number of states that have to be investigated within the search space. The results of both backtracking algorithms, and all modifications to each, are evaluated and compared in Sections 5.3 and 5.6 in order to determine the most effective automated approach.

Chapter 6 summarises the puzzle properties determined and offers conclusions on the knowledge of Kakuro gained. Suggestions are also presented for future work to extend knowledge of the underlying properties of Kakuro puzzles.

Chapter 2

Terminology & Literature Review

The terminology that will be used throughout this thesis to describe puzzle properties is now introduced. A review of the literature relating to Kakuro puzzles is then given. Since there exists limited material specifically related to Kakuro, work connected to similar puzzles and methods implemented for their solution are also considered.

2.1 Terminology

Let a Kakuro grid be termed K, where K has dimensions $n \times m$, and the cell entry at row i and column j be termed $k_{i,j}$. The top row and the left column of black cells are not usually included when describing the dimensions of a Kakuro grid. Each cell is either a white cell (to be later assigned a numerical value from a given, valid range, usually $1, \ldots, 9$) or a black cell. White cells within grid K belong to the set W and collectively form runs, each of which are exclusively either horizontal or vertical. Each run is represented as a tuple r_l $(l = 1, \ldots, p)$, where $r_l \in r$, the set of all tuples and p is the number of runs contained in the puzzle grid. We define the tuple r_l to be such that it contains no repeated elements. Most black cells contain numerical "clues" which are the run-totals $(t_l \in t)$ that correspond to the runs within the puzzle grid; the clues are placed adjacent to the vertical and horizontal runs.

Therefore r_l is described either as a tuple of connected horizontal white cells or of connected

vertical white cells. A horizontal run is defined:

$$r_l = (k_{i,j_s}, \ldots, k_{i,j_e}) \qquad k_{i,j_x} \in r_l,\ s \leq x \leq e$$

where the run is in row i $(1 \leq i \leq n)$, beginning in column j_s and ending in column j_e $(1 \leq j_s < j_e \leq m)$. A vertical run is defined:

$$r_l = (k_{i_s,j}, \ldots, k_{i_e,j}) \qquad k_{i_x,j} \in r_l,\ s \leq x \leq e$$

where the run is in column j $(1 \leq j \leq m)$, beginning in row i_s and ending in row i_e $(1 \leq i_s < i_e \leq n)$.

The cell entries, $k_{i,j}$, to be placed within the white cells are governed by puzzle constraints, namely that:

- In each and every run, $r_l \in r$, the same value must appear in no more than one cell entry:

$$k_{i,j} \neq k_{i',j'} \qquad \forall k_{i,j}, k_{i',j'} \in r_l$$

- In each and every run, $r_l \in r$, the corresponding run-total, t_l, must be satisfied:

$$\sum_{k_{i,j} \in r_l} k_{i,j} = t_l$$

A blank Kakuro grid becomes a *puzzle* when run-totals are added to the black cells, such that valid arrangements of values can be added to white cells to obtain a *solution*. Puzzles can be constructed such that there is always a move that can be made through the application of logic alone, such as there always being at least one cell in a Kakuro puzzle for which only one value assignment is legitimate. Most published puzzles conform to this convention [9, 35] and the current author adopts this viewpoint. Puzzles may also be constructed such that there are several candidate values plausible for the most constrained cell, requiring "guess-work", where the future repercussions of a possible placement are considered [36]. Although it is possible for puzzles to possess multiple solutions, a *proper*, well-formed puzzle should have a single, unique solution (which is especially important for those that are published).

Many published puzzles are pre-assigned a difficulty rating, with typical terms including "easy", "medium", "hard" or even "super hard". Such grading of puzzles can be based on many factors. Some grading practices may simply grade according to the presence of

longer runs, limiting the use of longer runs within easier puzzles. Others grade according to the possible number of arrangements of values that satisfy the run-total of a given run, with easier puzzles having mostly runs with few possible arrangements. Alternatively, puzzles that are automatically generated may be simultaneously graded by the same algorithm that is used to generate or solve the puzzle itself. Paul A. Grosse, a programmer who has generated thousands of puzzles in such a way commented that his program "...knew just how many times a particular algorithm was used to solve a particular puzzle, so these were graded and divided into appropriate groups of hardness" [26]. Such grading methods are rarely explained in detail so result in difficulty ratings that provide some indication of likely difficulty for a human solver without offering precise metrics for their calculation. Grading is examined in more detail in Section 3.1.1.

2.2 Literature Review & Survey of Methods

Very little work currently appears in the academic literature specifically on Kakuro puzzles, apart from work published by the current author [13, 14, 15]. Kakuro puzzles are known to be NP-complete. The field of complexity analysis divides problems into those that can be solved in polynomial time and those that cannot be solved in polynomial time, no matter what algorithm is used. Polynomial time refers to the number of computation steps a computer or an abstract machine requires to evaluate the algorithm; that is the running time of an algorithm [55]. An example of a problem that can be solved in polynomial time is a quicksort sorting algorithm on n integers, which performs at most An^2 operations for some constant A: thus it runs in time $\mathrm{O}(n^2)$ [11]. Some problems may be more complex and so may be of order $O(n^k)$ where k may be large or may even be an exponential function such as $O(e^n)$. A nondeterministic problem, one where there exists one or more situations where there are choices and different routes possible, that is verifiable in polynomial time is said to be NP and is *NP-complete* if any NP-problem can be translated into it. Possible solutions to NP-complete problems can be verified in polynomial time but there is no known efficient way to locate a solution in the first place. If a general algorithm was found to solve one NP-complete problem, then *all* NP-complete puzzles could be solved using such an algorithm. In terms of Kakuro, although valid solutions to a given puzzle can be verified easily in polynomial time (by examining whether puzzle constraints are satisfied by the current placement of values), no general algorithm is known for finding their solution. To prove that

a given puzzle or problem is in fact an NP-complete problem, it is sufficient to show that an already known NP-complete problem reduces to such a puzzle. The NP-completeness of Kakuro puzzles has been proven [56] by the reduction to a Kakuro puzzle of the well known Hamiltonian Path Problem [6, 67], and 3SAT [41, 51, 67], the restriction of the Boolean satisfiability problem, both known to be NP-complete.

In this section, emphasis will be placed on work published on similar puzzles and the potential applicability of the work to Kakuro puzzles. Brief introductions to the rules of such puzzles are also provided.

2.2.1 Crossword Puzzles

2.2.1.1 British Style Crossword Puzzles

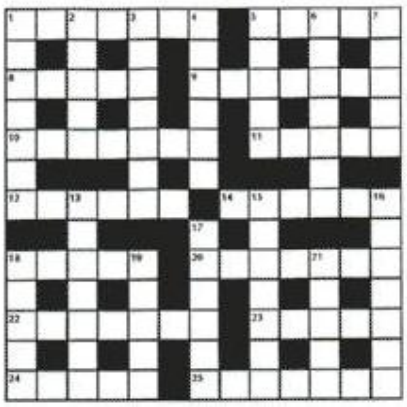

Figure 2.1: A British style Crossword [1]

The structure of British style Crosswords (Fig. 2.1) closely resemble Kakuro puzzles with respect to the structure of their puzzle grids. This puzzle consists of a $n \times m$ grid containing black and white cells. White cells collectively make word slots that are exclusively either horizontal or vertical, like the runs of a Kakuro puzzle. The aim of a Crossword puzzle is to place a word, which must appear in the dictionary that is currently in use, into each word slot, such that each and every white cell contains an alphabetic symbol from the valid alphabet currently in use. Black cells remain unused. Words are not usually used more than once and must be derived by solving an associated clue, usually either being of a "standard, quick" variety or a more complex "cryptic" variety. Not all cells within this type of cross-

word puzzle are interlocking; some are members of one word-slot, others of two word-slots. A constraint based approach toward the automated solution of this type of crossword puzzle has been implemented by Wilson [68] and is outlined in Section 2.2.1.2.

2.2.1.2 Fully Interlocked Crossword Puzzles

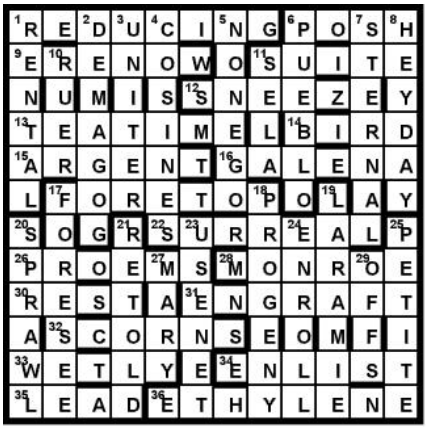

Figure 2.2: A fully interlocked Crossword [63]

An alternative type of Crossword puzzle, (Fig. 2.2), again comprises of a square or rectangular grid, but does not contain any black cells. This type of puzzle is said to be *fully interlocked* and sometimes contain "walls", which break up a row or column into multiple word slots. Alternatively, a word slot can be the entire length of a row or column. Words are derived by solving an associated clue, like when solving a British Style crossword (Section 2.2.1.1).

Constraint based approaches view a given problem as a collection of variables whose values must be assigned such that various stated constraints are satisfied. Constraint based approaches are explained in greater detail in Section 4.4. Binary integer programming (B.I.P.), a specific type of constraint based approach, has been used to attempt to solve a specific $m \times m$ fully interlocked Crossword puzzle grid with $n = 2m$ word slots, where $m = 4$ [68]. Two methods were investigated; Whole-Word insertion (a) and Letter-by-Letter insertion (b). The former method uses the following definitions of sets:

- I is the set of cells ($i \in I$),

- J is the set of letters in an alphabet ($j \in J$),
- K is the set of words in the lexicon ($k \in K$),
- N is the set of available word slots ($n \in N$).

Binary variables, $z_{n,k}$, are defined such that:

$z_{n,k} = 1$ if k is placed in slot n
$z_{n,k} = 0$ otherwise

Constraints are explicitly stated:
A word k can either appear once in word slot n or not at all;

- $\sum_{n \in N} z_{n,k} \leq 1 \qquad \forall k \in K,$

Each word slot n must be occupied by some word k from K;

- $\sum_{k \in K} z_{n,k} = 1 \qquad \forall n \in N,$
- Constraints to govern the set of words that can be allocated, given the word slot intersections were also added to the model.

The approach of Wilson [68] requires $2km$ variables with $2m + k + 26m^2$ constraints (where $2m = N$), when a lexicon of k m-letter words is used. Integer programming approaches, like other constraint based approaches (Section 4.4), typically find the optimal solution to a given problem, so an objective function is required. Typically, such a function would state that some quantity within the model is maximised or minimised so that the solution found is the "best" solution. Only a single solution to the crossword puzzle is required, so a dummy objective function is instead used to guide the solver, since any state that satisfies all the constraints is a desired solution state. The results of Wilson show that the approach found some solutions but not in a time that was satisfactory given the relatively small grid-size (but within 3,000 CPU seconds). The storage of the lexicon is problematic since, for this approach, it would essentially need to be stored multiple times since each word slot uses information on each word of the lexicon. This overall lexicon size would present a task for an integer programming model that the original authors described as "almost impossible" [68] . Wilson acknowledges that research is active in developing a hybrid approach for logical problems [4], but re-emphasizes his aim to use only commercially available software.

The paper concludes that hybrid logic/integer programming software is unlikely to become commercially available for some considerable time, if at all, and the prospects of using such approach with a realistic sized Crossword compilation is remote.

The latter approach, (b) a letter-by-letter approach, requires fewer variables and constraints, which are dependent on the number of rows and columns in the puzzle grid rather than on the size of the lexicon. This time, the following sets are defined:

- I is the set of rows in the crossword ($i \in I$),
- M is the set of columns in the crossword ($m \in M$),
- J is the set of letters of the alphabet ($j \in J$).

Binary variables, $x_{i,m,j}$, are defined such that:

$x_{i,m,j} = 1$ if letter j is placed in the cell in row i and column m
$x_{i,m,j} = 0$ otherwise

Constraints are explicitly stated:
Each cell must be occupied;

- $\sum_{j \in J} x_{i,m,j} = 1 \qquad \forall i \in I, m \in M,$
- For each cell and each letter in the alphabet, constraints that state the set of letters that by virtue of the lexicon could be placed into the cell are also developed.

The approach requires $26n^2$ variables with $27n^2$ constraints, where $n = |I| = |M|$ is constant, so is independent of the lexicon size. As puzzles of this sort do not, for example, have to maximise a score or minimise an error, a dummy objective function is again used to guide the solver. Only the assignment of values to variables that satisfy all constraints is desired. Despite appearing promising, the approach failed to reach a solution within 3,000 CPU seconds, the time required for the first method to reach a solution to the specific 4×4 grid under consideration. In addition, Wilson states that not all solutions, using this method, would contain words that are contained in the lexicon because it is "too loosely logically constrained". All crossword solutions are also solutions to the model, but not vice-versa. This approach is an attempt to show that formulations do exist that are not determined by

the size of the lexicon; such formulations are reported as being "impractical" [68].

Approach (a), word-by-word insertion, was also used for the attempted automated solution of a British Style crossword grid (similar to that of Fig 2.1); Wilson states that these puzzles appear more favourable than their "fully interlocked" counterparts in terms of the numbers of variables and constraints required compared to this fully interlocked counterpart [68]. However, even with a modest sized lexicon, $12k$ variables are required (where k is the number of words within the lexicon) which still constitutes a large number. Wilson concludes that the prospects of using B.I.P. for solving any puzzle of realistic size with a substantial lexicon is bleak. Wilson [68] concludes that integer programming approaches which are better able to take advantage of logical elements in combinatorial models would be far better suited to this type of problem, when, or if, they are developed. B.I.P. is investigated in relation to Kakuro puzzles in Section 4.4. Since Kakuro puzzles possess explicit numerical constraints and do not use a lexicon, the validity of "words" placed within the grid can be evaluated using run-total and non-duplication constraints alone. This approach is promising for providing an automated method for the solution of such puzzles.

2.2.1.3 American Style Crossword Puzzles

Figure 2.3: An American style Crossword [2]

American Style Crossword puzzles (Fig. 2.3) are similar to their British counterparts, having black cells that divide word slots, except that every white cell is interlocked; that is, every white cell is a member of two word slots.

An approach toward the automated solution of this type of puzzle was to investigate whether they could be solved as Probabilistic Constraint Satisfaction Problems [38]. As described above, a constraint satisfaction problem seeks a solution to a given problem such that all declared constraints are satisfied. They generally have no notion of a "best" or "worst" solution. Probabilistic constraint satisfaction problems extend the idea of a general constraint satisfaction problem by assigning a scoring mechanism to possible solutions so that a notion of a "better" or "worse" solution is introduced. Specifically, a solution where words actually match the given clues is desired rather than one which simply fills the grid with random interlocking words. Littman *et al.* [38] represent the problem as a collection of variables (word slots) and explicitly stated constraints that ensure that the "across" and "down" words interlock correctly. Formally, the problem is defined as the set of n variables, x_i:

$$X = \{x_i \mid 1 \leq i \leq n\}$$

each with domain D_i, representing a set of candidate answers that can be added to the variable (word slot), all of which are of the required length and seem to match the clue. Littman *et al.* [38] identified possible shortcomings of this: limiting domains to small sets may exclude critical candidates by oversight or due to the natural ambiguity of natural language. Conversely, over-generating candidate sets may allow erroneous solutions to be generated.

To attempt to combat the above problems (the exclusion of critical candidates and the generation of erroneous solutions), Littman *et al.* [38] assign a probabilistic preference to each variable. These give the solver a way to "rank" different possible solutions ("better", "worse" *etc*). During solution, variables are *coupled* through a constraint proposition *match*, which is defined on pairs of variables and possible values. For example, if x_i and x_j are variables and v and w are values, the proposition is true if and only if the partial instantiation $\{x_i = v, x_j = w\}$ causes no constraint violations. Each pair of variables, and all such *match* relations are stored in a *constraint network*. The probabilistic preferences information is then added to the network in the form of probability distributions over domains, *i.e.* $p_{x_i}(v)$ is the probability that $v \in D_i$ is present in variable x_i. The sum of all such probabilities must be 1.

The solutions are generated by selecting a value for each variable, depending on their prob-

ability distribution p. These values are "kept" if constraints are satisfied or discarded and replaced otherwise. Therefore, the probability of a solution is then proportional to the product of these probabilities, normalized by the total probability assigned to valid solutions. The algorithm scored 89.5% of words correct for a sample set of test puzzles, each defined as "challenging". Littman *et al.* [38] concluded that combinations of constraint satisfaction and probability theory hold promise for attacking this and a wide array of other problems.

"Words" that fill the runs of Kakuro puzzles do not come from a lexicon as such and the success or failure of a placement should be evaluated using the run-total and non-duplication constraints alone. However, a particular arrangement of values may be more suitable for placement into a given run because of the run-total(s) belonging to the intersecting adjacent runs. Probabilistic preferences could therefore be added to distinguish between different value arrangements that sum to an equal run-total. The size of a domain, D_i, that corresponds to a run ("word slot") may be very large, particularly for long runs (Section 3.1.2). This is in stark contrast to the domain of a word slot within a Crossword puzzle, in which the domain will tend to decrease as the word slot length increases. A word clue is generally more constraining than a run-total; a number sequence may be permuted into more legitimate forms than a letter sequence. Hence this method has not been pursued.

2.2.1.4 Go-Words Puzzles

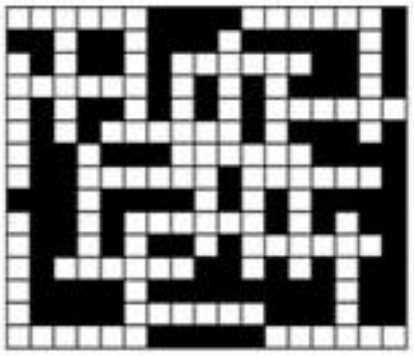

Figure 2.4: A Go-Words puzzle

Go-Words puzzles consist of an empty crossword grid where all word slots within the puzzle grid are the same length. Unlike the variations above, there are no clues associated with the

word-slots and each letter is assigned a score. The aim of the puzzle is to place words (all of length six in the example of Fig. 2.4) into the word slots such that the words intersect correctly. A lexicon of words, all of the correct length, is provided and contains more words than there are word slots within the grid. Once a solution is found, a total score for the grid can be calculated using the individual letter scores given, aggregating the scores for all letters in the grid. The score attached to a particular letter varies from puzzle to puzzle so the player would aim to achieve the highest score possible from a valid arrangement of words within the specific grid.

A genetic algorithm approach for the automated solution of this Crossword puzzle variant is given by Purdin and Harris in [50]. A genetic algorithm is a metaheuristic search technique. They are a very general algorithm and so will work well in any search space [60]. Genetic algorithms are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as mutation, inheritance, selection, and crossover. (Genetic algorithms are explained in greater detail in Section 4.3.) Such algorithms represent the puzzle (or incomplete puzzle) as a bit string called a "chromosome", the actual representation of which was chosen to be a letter string, where letters at intersections appear only once. Purdin and Harris note that alternatively, a pure binary representation or a letter by letter representation could also have been chosen.

An *initial* population is established by inserting random words, horizontally before vertically, from the lexicon; this is "a good start, but not a perfect start". As the algorithm progresses, an objective function determines the "fitness" of each of the chromosomes by assigning some score. Such a function would need to ensure all words in a solution appear in the lexicon and would also need to differentiate between multiple valid solutions, based on the Go-Words letter scores provided. The algorithm of Purdin and Harris uses crossover and mutation operations to form new generations within the population. Crossover operations aim to maintain words where they exist and to promote them where they do not. Mutation operations aim to continually introduce "new" information into the population. Ensuring that such damage is minimised is accomplished by fine parameter tuning [50].

Purdin and Harris note that the algorithm performs well, but not as well as a human solver, possibly due to overheads in chromosome manipulation. Key issues involve the introduction of new information without introducing extraneous mutation. Purdin and Harris believe

that the genetic algorithm could be further refined to take into account more puzzle-specific information that may be useful in reaching a solution more quickly. Alternatively, initial populations could also be "seeded" with known, good solutions to maximise the effect of the algorithm.

Due to the numerical nature of Kakuro puzzles, this approach may be effective toward their automated solution. Since the current validity of values placed within the grid can be determined by the current run-totals and whether violations have occurred, determining the "fitness" of each of the chromosomes (by assigning some score) may be easier than for those related to the Go-Words grid. The numerical nature of a Kakuro puzzle, specifically the run-totals, suggest that an appropriate fitness function could be generated. Crossover and mutation operations would be required to fill runs with "number words" using values from the standard numerical range rather than by searching a lexicon of limited size. This increase in possible assignments, and the difficulties of determining suitable crossover and mutation operations may make genetic algorithms even less efficient for Kakuro puzzles. Nevertheless, an application is considered in Section 4.3.1.

2.2.1.5 Unconstrained Crossword Puzzles

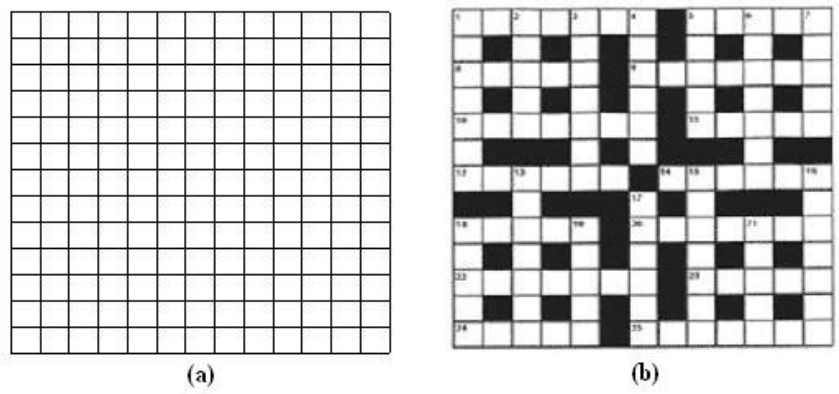

Figure 2.5: An initial unconstrained grid (a) and an example puzzle geometry (b)

An *unconstrained* crossword puzzle is a variation in which only the grid dimensions and the corresponding lexicon are known. Such puzzles are initially only an $n \times m$ grid containing white cells (as in Fig. 2.5(a)). A solving algorithm would therefore need to determine the

locations of the black cells, the word slots and the words that fill such slots. Like a Go-Words puzzle (Section 2.2.1.4), there are no clues associated to the word-slots; words of the correct length that appear in the given lexicon must merely fill word-slots, obeying intersection rules. Differing arrangements of word slots and black cells are collectively known as the puzzle geometries. Puzzle geometries are crucial so should be considered as part of a solution together with the words, from the corresponding lexicon, that are to be placed in word slots. One such puzzle geometry is shown in Fig. 2.5(b). Many puzzle geometries are trivial (for example, grids with one word-slot or grids filled with black cells). An unconstrained problem can therefore be thought of as a generalisation of the constrained problem. In principle, a successful solver for constrained problems [38, 50, 68] can be executed for each puzzle geometry to obtain an entire solution set, however, such an approach would be highly time consuming since the solver would have to be run 2^{nm} times; once for each puzzle geometry that exists for an $n \times m$ grid [29].

Any attempt to solve unconstrained Crossword puzzles would need to eliminate trivial geometries but since it is not clear on what basis to remove these, this approach would not be practical. Instead, an approach where black cells are filled only as the algorithm progresses to a solution is desired. Such an approach would ignore trivial geometries while efficiency would depend on dictionary size. Recursion and backtracking is used in [29] to fill a letter slot table where, in turn, attempts are made to add another word to the table that intersects with the letter under consideration. Initially, a word is chosen from the lexicon or can be user-defined and is placed in the centre of the empty grid. If successful, or if no word can be entered but further letter slots still exist, the next letter slot is considered. Otherwise, a failure has occurred, causing backtracking. In contrast to the constrained problem, not every letter slot needs to be filled for a solution to be valid, with empty cells becoming black. A "density" of black squares is compared to a threshold to ensure that there is a sensible number of black cells present. Harris [29] concludes that it is still time consuming for the algorithm to generate a complete solution set for a given dictionary, so an objective function could be used to distinguish between "good" and "poor" solutions.

This approach is not applicable to Kakuro puzzles since the grid geometries, the placement of white cells and black cells, are always provided.

2.2.2 Sudoku and Rodoku Puzzles

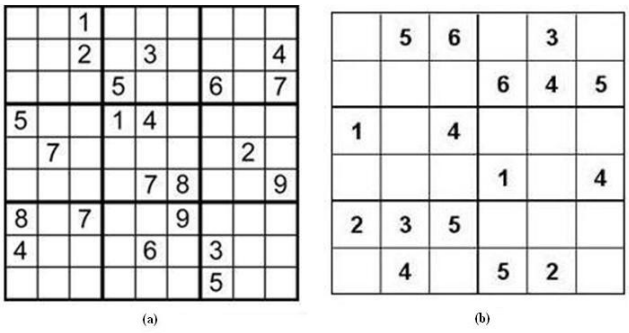

Figure 2.6: Sudoku and Rodoku puzzles [65]

Standard Sudoku puzzles (Fig. 2.6(a)) are composed of a 9×9 grid, itself divided into nine 3×3 mini-grids. Each row, column and mini-grid must contain the values $1, \ldots, 9$ once, with a *well-formed* Sudoku grid having a unique solution. The structure of Sudoku puzzles has been shown to be useful for the solution of several real-world problems (Section 1.2). Unlike Kakuro puzzles, Sudoku Puzzles contain *givens*; values that are pre-placed in some cells, chosen to ensure a solution is unique. There is an extremely large number of goal state solutions; 6,670,903,752,021,072,936,960 in total [18]. This class of puzzles has been shown to be NP-complete for higher dimensions [70].

For every non-prime dimension n, there is an $n \times n$ Sudoku grid [27]. However, not every size of Sudoku grid will contain $m \times m$ mini-grids for some integer m. As an example, a 6×6 grid (known as a Rodoku puzzle [shown in Fig. 2.6(b)]) contains mini-grids of size 3×2 or 2×3 [34]. Sudoku puzzle grids, of size $n \times n$, always consist of n^2 white squares, each of which must contain a value in the range $1, \ldots, n$. It is always known beforehand what values require placement within each row, column and mini-grid within a Sudoku puzzle; only the ordering of such values is initially unknown. This is in contrast to a Kakuro puzzle, which can widely vary in structure from puzzle to puzzle. In a Kakuro puzzle, the overall grid size can vary widely between puzzles, neither the particular values to be placed into a run nor the order of such values are known beforehand, and there may exist black cells within the grid structure. Kakuro puzzles rely on both non-duplication and summation constraints whereas Sudoku and its variants rely only on non-duplication constraints. It is for these reasons that it is more difficult to generalise an approach to the automated solution

of Kakuro puzzles. (For example, it is not generally sufficient to simply allocate values to cells and then iteratively swap pairs of them.)

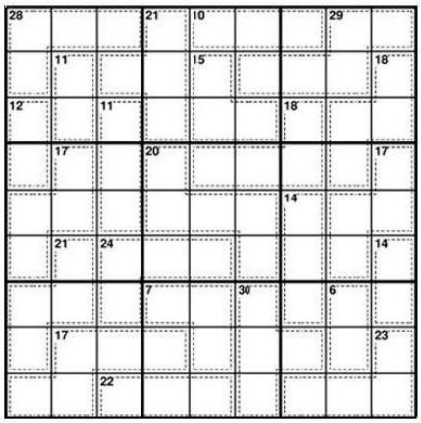

Figure 2.7: A Killer Sudoku

Killer Sudoku puzzles (Fig. 2.7) adopt the same rules as standard Sudoku puzzles. In addition, the 9×9 grid is further subdivided into cages. The size and shape of cages within a certain puzzle grid can differ greatly from those of a different puzzle, rather like the runs within a Kakuro puzzle grid. Each cage has an associated summation requirement and cannot contain duplicate values. Cages are generally at least two cells in size, although some published puzzles exist where a cage comprises of a single cell; this cage would be somewhat redundant because the cage total would instantly show what value is to be placed into the cell. This would be the equivalent to a given value in a standard Suduko. Unlike a standard Sudoku puzzle and depending on puzzle difficulty, a Killer Sudoku puzzle may or may not contain given values.

Automated solutions of Sudoku-type puzzles could reasonably be divided into two categories; *constraint based approaches* and *heuristic based optimization algorithms*. The former, which for example, include the use of flow algorithms or the application of bipartite matching [58], view the problem as a set of variables that require the assignment of a set of values. The approach can therefore effectively mimic the methods that may be used by a human solver and take advantage of puzzle constraints. Interest in such constraint based approaches may be due to the fact that Sudoku is very closely related to Latin Squares and hence, to the

much-studied *Quasi-group Completion Problem*. A quasi-group is a pair $(Q, *)$, where Q is a set and * is a binary operator such that $a * x = b$ and $y * a = b$ are uniquely solvable for every pair of elements a, b in Q. The multiplication table of its binary operation defines a Latin square (*i.e.* each element of Q appears exactly once in each row and column). Sudoku can therefore be thought of as a quasi-group completion problem with additional constraints (namely the requirement of distinct values in each mini-grid as well as each row and each column) [8]. Quasi-group Completion Problems have been linked to many real-world applications including experimental design [24], timetabling, error-correcting codes [3] and routing in fiber optic networks.

A local search approach to the automated solution of Sudoku puzzles, employing an objective function to guide the process of search toward a solution, has proved fairly successful [33]. A description of local search and objective functions is given in Section 4.2. A modified steepest ascent hill-climbing algorithm is detailed, which employs an objective function to move through the search space. Such an objective function indicates how promising a path seems to be. Backtracking (explained in Section 4.1) enables movement back from "dead end" paths, hence avoiding local optima in objective function scores. Jones *et al.* [33] acknowledge that without modification, this algorithm may become trapped in large plateaus in objective function scores, creating difficulties in deciding on which path to follow. Due to Kakuro sharing with Sudoku a non-duplication constraint, a local search approach to the automated solution of Kakuro puzzles is considered in Section 4.2.1, with the findings of that investigation being placed in the context of the results presented by Jones *et al.*

Gago-Vargas *et al.* [21] consider Sudoku puzzles as a graph colouring problem, where a puzzle is modelled as a graph with 81 vertices (one for each cell) requiring 9 colours (one for each number). Edges are defined by the adjacency relations of Sudoku relating to row, column and mini-grid constraints. The colouring problem is solved through a system of polynomials, described by an ideal [12] over the rational field, $I = \mathbb{Q}[x_1, \ldots, x_{81}]$ (one rational variable for each vertex). I is of the form $I = \{x_i - a_i | i = 1, \ldots, 81\}$, where a_i are numbers between 1 and 9. A Gröbner basis of an ideal is a particular type of generating subset which must be a non-zero polynomial. I has two generators, Gröbner bases F and G such that:

$F(x_j) = \prod_{i=1}^{9}(x_j - i) \qquad j = 1, \ldots, 81$

$G(x_i, x_j) = \frac{F(x_i)-F(x_j)}{x_i-x_j} \qquad 1 \leq i < j \leq 81$

Sudoku puzzles contain given values so these are added to the ideal, I. For example, if a cell represented by variable x_2 contains given value 2, the polynomial of the form $x_2 - 2$ is added to the ideal.

Since the placement of a value into a cell can affect twenty cells (those sharing a row, column or mini-grid), the graph has degree 20 and $\frac{81\times 20}{2} = 810$ edges. Since the "colours" 1 to 9 are being considered, the polynomial $\prod_{i=1}^{9}(x_j - i)$ is considered. If two vertices are adjacent, then $F(x_i) - F(x_j) = (x_i - x_j)G(x_i, x_j) = 0$, meaning that the condition about differing colours is governed by the polynomial G.

Gago-Vargas *et al.* found that following the addition of polynomials to I relating to the given values, the polynomials F are redundant so can be removed, since polynomial G alone governs the condition regarding differing adjacent colours. Therefore the system comprises of 810 equations, one for each edge. Providing a solution exists, the solutions of a given puzzle are contained in the set of zeroes of I which is:

$$\{(s_1, \ldots, s_{81}) \in \mathbb{Q}^{81} \text{ such that } H(s_1, \ldots, s_{81}) = 0 \text{ for any } H \in I\}$$

Hence, any reduced Gröbner basis of H in I will have the form $G = \{x_i - a_i | i = 1, \ldots, 81\}$ where every a_i is in the range $1, \ldots, 9$ and correspond to a solution. A solution is therefore said to be "encoded in a reduced Gröbner basis" [21].

Gago-Vargas *et al.* acknowledge that in general, the systems produced by Sudoku puzzles are not "friendly". Backtracking algorithms are much more widely used and can arrive at a solution quickly, whereas computing a Gröbner basis can be computationally expensive, making it a rather unattractive solution method. However, an advantage of this approach is that if a Sudoku grid is not well-formed, it is possible to obtain the number of solutions using this method. Interestingly, Gago-Vargas *et al.* proposed that this approach may be used with Kakuro puzzles by using the generators:

$F(x_j) = \prod_{i=1}^{9}(x_j - i)$ for each cell
$G(x_j, x_k) = \frac{F(x_j)-F(x_k)}{x_j-x_k}$ for cells (j, k) in the same run

Polynomials relating to run-total constraints are added to the ideal and should be of the form $x_1 + x_2 - 4$ where, for example, x_1 and x_2 are the variables corresponding to cells one and two, both of which are members of a run with corresponding run-total 4. Gago-Vargas *et al.* did not implement this approach. The automated solution of Kakuro puzzles is considered by the current author in Chapters 4 and 5. It is desired that methods for solution be relatively computationally efficient and extendible to grids of large size. Despite being a "powerful tool" [31] for the solution of a structure described by generators, the proposed Gröbner basis approach does not meet these requirements, since calculating a Gröbner basis is typically a very time-consuming process for large polynomial systems [66] and can be highly computationally expensive [31, 47].

2.2.3 Quasi-Magic Sudoku

						9		
		8	1				4	
5			7	2			8	3
1			5		2	8		
	5			3			7	
		9	8		7			2
9	1			4	8			6
	4				1	3		
		3						

Figure 2.8: A Quasi-Magic Sudoku with $\Delta = 2$ [20]

In addition to Rodoku puzzles (Fig. 2.6b), more variants of the standard Sudoku puzzle exist. Fig. 2.8 shows an example *Quasi-Magic Sudoku* puzzle with $\Delta = 2$.

The puzzle adopts all the rules of the standard puzzle but imposes an additional rule: each of the row, column and diagonal sums of the 3×3 mini-grids must be a number in the range $15 \pm \Delta$, where Δ is a fixed parameter. Like the standard puzzle, Quasi-Magic Sudoku puzzles contain given values, but the addition of these *Quasi-Magic constraints* reduces the number of givens required to ensure the uniqueness of solution. The case $\Delta = 9$ imposes no

additional constraints over an ordinary Sudoku puzzle, since all mini-grid rows, mini-grid columns and mini-grid diagonals add up to a total in the range $6, \ldots, 24$. For a standard 9×9 grid, it has also been shown that Δ cannot equal 0 or 1 [20]. Conditions on the legal placement of values in such puzzles are reported by Forbes [20] and Roach *et al.* [54] and subsequently proved by Jones *et al.* in [32]. These include:

1. The value 5 can only be placed in the centre cell, or in a corner cell, of any mini-grid,
2. At most, only one mini-grid will have 3 in its centre cell; the same applies for the value 7,
3. The values 6 and 7 can not form mini-grid centres in the same stack or band.

These conditions are used as pruning rules in an automated approach to the solution of Quasi-Magic Sudoku puzzles in [54], using a recursive backtracking depth-first search approach (Backtracking is explained in Section 4.1). This pruning reduces the amount of the search space that would have to be considered. In this implementation, cells are defined by two variables: a flag vector, denoting which values may currently still be assigned to that cell (*i.e.* the candidate values) and a "just fixed" flag, denoting that the content value of the cell has recently been fixed. The flag vector indicates the candidate values through the use of 9 consecutive bits - each bit representing a different value with all values initially available. A value is classed as non-assignable if the corresponding bit is zero. The assignment of a value to a cell results in bits corresponding to that value in the same row, column or mini-grid to be set to zero. This approach is known as bitmasking. (Bitmasking is explained in more detail in Section 5.5.1.) At each stage, the algorithm attempts to assign a value to the next cell to be considered, by selecting the numerically lowest of the remaining candidate values for that cell. During each iteration, Quasi-Magic pruning rules ensure that relevant cells are not assigned disallowed values.

The assignment of a value to a cell is indicated by the just fixed flag of the cell becoming set, resulting in violation checks taking place. This algorithm returns a Boolean value indicating whether an assignment is valid, and also fixes cells that have indirectly been limited to a single candidate value. A grid is rejected if any cell is reduced to having no remaining candidate values, as this makes the grid impossible to complete. A cell-ordering heuristic is also employed; rather than considering cells in some consecutive order in terms of their physical

position in the grid, they are instead ordered according to how few remaining candidate values they possess. Roach *et al.* [54] conclude that the solver that uses the cell ordering heuristic with effective pruning rules was most successful. The majority of puzzles require less than 0.1 seconds for solution with the worst case requiring 1.2986 seconds.

It has been proposed that Sudoku may be useful for the construction of erasure correcting codes [62]. A message to be sent is encoded into the structure of a Sudoku puzzle. Information lost in transmission is recovered by solving the puzzle, where correctly received values are akin to puzzle givens. In a general sense, Roach *et al.* [54] believe their above findings establish the applicability of Quasi-Magic Sudoku for the construction of erasure correction codes, since Quasi-Magic Sudoku carries the prospect of grid reconstruction (and hence message reconstruction) from a smaller set of initial given values. However, they report that to prove the usefulness of Quasi-Magic Sudoku in erasure correction, it is necessary also to consider reconstruction from a set of values which may not correspond well to the minimal set of independent givens of a particular grid [54].

The use in the above approach of a flag vector, showing which values are available for placement into a cell, may be beneficial in the implementation of an automated approach for the solution of Kakuro puzzles. If a certain value becomes unavailable for placement into the cell, the "flag" may be dynamically updated, providing an up-to-date indication of available values and avoiding the exploration of "dead ends" within the search space (Section 4.1). This approach has been implemented in Section 5.5.1.

2.2.4 Survo Puzzles

	6			30
8				18
		3		30
27	16	10	25	

Figure 2.9: A Survo puzzle grid [45]

Survo puzzles are an alternative type of cross-sum puzzle [45]. Survo puzzles are played on an open, rectangular grid and are not limited to integers from the range $1, \ldots, 9$. Instead, the aim of a Survo puzzle is to fill an $n \times m$ grid with integer values from the range $1, \ldots, nm$ such that each of the values appears only once and their row and column sums are equal to the *marginal sums*, integers given on the bottom and the right side of the table.

In a similar way to Sudoku puzzle grids, Survo puzzles can contain given values, chosen in order to guarantee uniqueness of solution. Fig. 2.9 shows a Survo puzzle with three such givens. If no givens are provided, a puzzle is an *open* Survo puzzle. Additionally, to ensure uniqueness of solution, all row sums must be different otherwise rows may be swapped. A similar constraint exists for column sums.

Mustonen [45] presents an upper bound on $S(n, m)$, the number of essentially different Survo puzzle grids for an $n \times m$ grid. The number of possible value arrangements, ignoring the marginal sums, is a poor upper bound. Instead, the number of possible sum partitions is used. Two open Survo puzzles A and B are defined as being essentially different if the solution of A cannot be transformed into the solution of B by interchanging rows and columns or by permutation of the values $1, \ldots, nm$. For example, a 3×4 puzzle grid may be filled using any of $12! = 479,001,600$ arrangements of values; however, many of these arrangements would not be essentially different. One grid may contain the same arrangement of values as another, obtained by using value permutation, row swapping or column swapping. Instead, the product is calculated of the number of ways that the sum of all values, $\sum_{x=1}^{mn} x$, may be partitioned, firstly into m distinct parts and secondly into n distinct parts. For the grid of Fig 2.9, a 3×4 puzzle grid, $\sum_{x=1}^{mn} x = 78$; this number may be partitioned into three parts in 128 distinct ways and may be partitioned into four parts in 519 distinct ways. The upper bound on $S(3, 4)$, the number of possible essentially different grid arrangements is therefore reduced from $12! = 479,001,600$ to $128 \times 519 = 66,432$. Mustonen [45] then uses a partially randomized swapping algorithm, implemented in SUMMAT, to find that the actual number of essentially different Survo puzzle grids, $S(3, 4)$, is 583. Mustonen reports an upper bound of 4,438,710 for $S(4, 4)$, and an actual number of essentially different grids of $S(4, 4) = 5327$ [45]. No larger grids are investigated.

The runs of a Kakuro puzzle are similar to the rows and columns of an open Survo puzzle.

A generating function for the total number of valid, unordered arrangements (partitions) of values within runs of a given size for a specified run-total is derived in Section 3.3, where the sum of placed values meet the specified run-total. Each partition can be permuted into different orderings, so the number of unordered arrangements determined by the generating function must be multiplied by the factorial of the number of cells present within the run. The number of ordered arrangements (compositions) of values within each run of a given size and total is then known. This generating function has been used to develop a look-up table that is employed in a heuristic during the automated solution of Kakuro puzzles in Section 5.2.1.

Mustonen intends that Survo puzzles be solved by human solvers employing logical reasoning, but also presents a computational approach [45], while noting that the automated approach does not employ that same logical reasoning. In this approach, the grid is initially randomly filled. Values are then swapped, step by step, until a solution is reached. This computational method is improved by use of a greedy algorithm to produce the initial grid arrangement. This algorithm places within each cell the value considered most likely by taking into account the product of the marginal sums of the row and column in which the cell resides. Small numbers are generally located in crossings of small sums and large numbers in crossings of large ones. So for each cell, the product of its horizontal sum and vertical sum are computed then assigned a value in the range $1, \ldots, nm$, based on the ranking order of the products obtained. The highest product is assigned a rank nm and the lowest is assigned a rank 1. Only then, following this initial ranking, did the swapping process begin, depending on how these sums deviate from the true sums. Mustonen [46] notes that this method does not ensure the uniqueness of the solution of non-open grids because the given values are ignored.

Kakuro puzzle grids may contain duplicated values (provided they are placed in distinct runs), so this method of ranking values cannot be used. However, the value to be placed into the cell located at the intersection of two runs within Kakuro puzzle grid may be similarly indicated by the two corresponding run-totals of those two runs. A cell placed at the intersection of two runs with high totals is likely to require the placement of a high value. A value ordering heuristic, explained in Section 5.2.3, calculates an "average" score for each cell within the puzzle grid, based on associated run-totals. A high average suggests that a high value is likely to be required while a low average suggests that a low value is likely to

be required. The idea of ranking is also considered in the context of the order in which cells of a Kakuro puzzle are filled; the use of the number of ways in which runs may be completed is considered in Section 5.2.1, and the use of the number of values that may be placed in an individual cell is considered in Section 5.5.2.

2.2.5 Automated Solution of Kakuro Puzzles

As mentioned previously in this section, very little work currently appears in academic literature that is specifically about Kakuro puzzles. However, Kakuro puzzles have been modelled as a constraint problem [59]. This method uses similar ideas to those first applied to crossword puzzles, as detailed in Section 2.2.1.2 and in greater detail in Section 4.4. Each cell is assigned a finite domain variable with values in the range 1 to 9. All puzzles are defined by a tuple $< G, H >$, where G is the set of all cell locations and H, itself a tuple $< v, I >$, represents the runs. H contains information about v, the associated run-total, and I, the set of cell locations to which v corresponds, such that $I \subset G$. The overall model therefore comprises of variables, x_i, for each cell i such that:

$$\forall i \in G : x_i \in [1, \ldots, 9]$$

A built-in "Search" procedure from the "propia" library of ECLiPSe [64] is used, with its default values and combined use of an *alldifferent* constraint, specifying that all variables x_i whose locations are stated in a given hint are pairwise different:

$$\forall < I, v > \in H : \text{ alldifferent } (\{x_i | i \in I\})$$

and a *sum* constraint, specifying that the sum of variables, whose locations are stated in a given hint, must meet the given run-total:

$$\sum_{i \in I} x_i = v \qquad \forall < v, I > \in H :$$

These constraints from the global constraint catalogue [5], are used with a search routine to find a solution matching puzzle constraints. Simonis [59] notes that since the aim is to automate the solution of puzzles by using constraint-based methods as opposed to search methods, this search routine serves only as a backdrop, to be used if the constraint-based approach fails.

With a timeout of 300 seconds imposed, none of the puzzles present in the test set (with 313 puzzles of varying grid-sizes between 9×9 and 124×90) were solved in this time without the initial use of a recursive *shaving* technique. Such a technique reduced the domain size of each variable by eliminating elements that could not possibly appear in a solution. Following the use of such shaving, 80% of puzzles were solved. When the backup search routine was used, over 98% of the puzzles were solved within the given time limit. Simonis [59] also found that by using a greedy algorithm that removed run-totals until multiple solutions appeared, a significant number of hints could be removed without losing the uniqueness of the solution, hence generating more challenging puzzles.

Since Kakuro puzzles possess explicit puzzle constraints that can be used to find a solution whose validity can easily be checked, Kakuro puzzles are seemingly appropriate for this constraint-based approach to a solution. This approach is addressed, using binary integer programming, in Section 4.4.

Chapter 3

Puzzle Grading and Valid Run & Grid Enumerations

In this chapter, the method by which puzzles are assigned difficulty gradings, such as "easy", "hard" *etc.*, is investigated in order to ascertain whether run properties are used alone, or in conjunction with less obvious methods, such as the use of computer-based grid production algorithms. Bounds on the number of valid arrangements of values that can be placed within given grids are then considered. Such an analysis examines the underlying puzzle properties that, where possible, will be later used to inform automated approaches to puzzle solution.

3.1 Initial Findings on Puzzle Complexity

3.1.1 Puzzle Grading

Many published puzzles are pre-assigned a difficulty rating, with typical terms including "easy", "medium", "hard" or even "super hard". Such grading of puzzles can be based on many factors. Some grading practices may simply grade according to the presence of longer runs, limiting the use of longer runs within easier puzzles. Others grade according to the possible number of arrangements of values satisfying run-totals, with easier puzzles having mostly runs with few possible arrangements. Alternatively, puzzles that are automatically generated may be simultaneously graded by the same algorithm that is used to generate or solve the puzzle itself. Paul A. Grosse, a programmer who has generated thousands of

puzzles in such a way commented that his program "... knew just how many times a particular algorithm was used to solve a particular puzzle, so these were graded and divided into appropriate groups of hardness" [26]. Such grading methods provide some indication of likely difficulty for a human solver but are rarely explained in detail. They rarely offer precise metrics for their calculation.

To investigate whether published puzzles employ a trivial method of puzzle grading, as opposed to a more complex method such as by internal algorithmic grading, forty puzzles from each difficulty rating are analysed here. Puzzles are taken from each of three difficulty ratings, "easy", "medium" and "hard". Each puzzle is analysed with respect to the run-lengths present and the number of different, ordered ways that each given run could be satisfied. For each difficulty grading, twenty puzzles are taken from over 900 9×9 puzzles from a specialist, puzzle book [10], source A in the tables below. The remaining twenty (also of size 9×9) are from a website [28], source B in the tables below, which provides over 600,000 puzzles of varying size with varying difficulty ratings. Hence, 120 puzzle grids are analysed. All puzzles contain runs which consist of at least two cells.

Table 3.1: Run-lengths present in the test set of puzzles

Rating	Source	Run-Lengths Present (% of total)								Average Length
		9	8	7	6	5	4	3	2	
Easy	Overall	0.00	0.00	0.41	0.54	5.84	18.61	23.91	50.68	2.83
	A	0.00	0.00	0.00	0.00	9.97	25.61	29.11	35.31	3.10
	B	0.00	0.00	0.82	1.10	1.64	11.51	18.63	66.30	2.55
Medium	Overall	0.00	0.15	2.47	4.65	6.84	16.29	22.11	47.49	3.07
	A	0.00	0.00	1.90	5.69	8.40	22.22	24.93	36.86	3.27
	B	0.00	0.31	3.14	3.45	5.02	9.42	18.84	59.81	2.84
Hard	Overall	1.20	3.01	2.56	7.38	4.52	14.01	21.84	45.48	3.32
	A	1.17	2.33	2.33	11.66	6.12	20.41	23.62	32.36	3.63
	B	1.25	3.74	2.80	2.80	2.80	7.17	19.94	59.50	2.99

Table 3.1 shows the percentage of the total number of runs which are of each possible length in puzzles from each source and for each difficulty rating, and the average length of runs. Puzzles graded "easy" from both sources appear to avoid the use of the longest possible runs (runs of length eight or nine cells). In fact, Source A [10] avoids the use of any runs

which are of length six cells or more, suggesting that these puzzles may be graded using run-length alone. This conjecture is reinforced by the fact that two out of the four "missing" run-lengths are reintroduced within puzzles that have the next highest grading, that of "medium". All run-lengths are then present in the hardest puzzles examined. As expected, the highest proportion of runs in all puzzles examined were of length 2 (the shortest possible length) which are most easily placed within the constraints of the puzzle grid's dimensions. As difficulty increased, this length continued to be the most frequent, but sees a decrease in its overall percentage share, as more grid space is taken up by longer, potentially more difficult runs. Overall, for both sources and for the test set as a whole, the average run-length increases, possibly suggesting that run-length, if not the overall deciding factor, is at least taken into account as puzzles were graded. Puzzles from Source B [28] do not seem to be so obviously differentiated using run-length alone, as longer lengths of runs are included in puzzles of lower difficulty ratings.

Table 3.2: Average number of permutations that can satisfy runs of given length

		Average Number of Permutations for Each Run-Length							
Rating	Source	2	3	4	5	6	7	8	9
Easy	Overall	3.761	22.653	117.372	643.256	1,800.000	17,640.000	n.a.	n.a.
	A	3.641	24.306	**119.495**	630.811	n.a.	n.a.	n.a.	n.a.
	B	3.826	20.029	112.571	**720.000**	1,800.000	17,640.000	n.a.	n.a.
Hard	Overall	4.480	23.628	117.742	646.000	2,813.878	13,637.647	40,320.000	362,880.000
	A	3.910	21.074	**114.171**	664.714	2,772.000	12,285.000	40,320.000	362,880.000
	B	4.812	26.859	120.522	**633.333**	3,000.000	14,840.000	40,320.000	362,880.000

Runs having the same length may be differentiated by the number of ordered arrangements of values that may be used to satisfy their run-totals. In Table 3.2, the average number of valid, ordered ways of satisfying runs of specified lengths are examined for "easy" and "hard" rated puzzles. With few exceptions, shown in bold, harder puzzles seem to contain runs that, on average, could accept more valid arrangements of values than their easier counterparts. It was earlier observed that runs of length two are the most frequent in all puzzles. This additional analysis now shows that despite the high presence of such runs in all puzzles of all difficulties, those within harder puzzles do indeed accept more arrangements of values, suggesting that more reasoning may be required for their solution.

3.1.2 Possible Run Arrangements

The total number of valid, unordered arrangements of values that may be placed in a run of any size, meeting any possible run-total can be derived by considering the number of valid binary strings [26].

Table 3.3: Binary representations of three example runs

Possible Cell Value	1	2	3	4	5	6	7	8	9	**Resulting Binary String**
a) 1 and 6 in a run of length two	1	0	0	0	0	1	0	0	0	100001000
b) 2, 4 and 6 in a run of length three	0	1	0	1	0	1	0	0	0	010101000
c) All values in a run of length nine	1	1	1	1	1	1	1	1	1	111111111

Each bit within a binary string represents a value (from the standard range $1, \ldots, 9$) that is or is not present in a run. Therefore, the length of the run corresponds to the number of bits within the binary string that are set equal to "one". So in the first example in Table 3.3, there is a 1 and a 6 present, meaning there is a binary "one" in the corresponding bits in the binary word (the first and sixth bit from the left). Conversely, the other seven bits are set equal to "zero", since the corresponding values do not appear in the run. This corresponds to a run of length two (because two values are present) with a run-total of seven. Two further examples are also given in Table 3.3.

The total number of such binary strings is therefore $2^9 = 512$. However, a run must consist of at least one cell so the zero-cell case must be discounted, leaving 511 valid sets of values. If, as is the case with the majority of published puzzles, a run must be at least two cells in length, all cases where the run-length is of unit length must also be discounted, reducing this figure further to 502. Of course, the arrangement of values within runs is also of great importance. For example, although there is only one set of values that can be used to satisfy a run-total of 45 (namely all values $1, \ldots, 9$), these values can be ordered in $9! = 362,880$ ways. Although there are only 502 sets of values that can be used to fill runs of all possible run-lengths that sum to all possible run-totals, when the alternative orderings of these sets of values is taken into account, there are $986,400$ different orderings.

3.1.3 Solution Uniqueness

While investigating the automated solution of Kakuro puzzles using constraint based approaches, Simonis [59] (detailed in Section 2.2.5) found that between 3% and 16% of the overall number of run-totals can be removed from puzzle grids within his test set without losing the uniqueness of a solution, possibly generating more challenging puzzles due to the lesser number of such clues present.

When run-totals are progressively added to an empty grid in order to form a puzzle grid, only one arrangement of values should form a valid solution, assuming the puzzle is well-formed. However, in solving a puzzle (either manually, or automatically), it is inevitable that multiple arrangements of values in runs must be considered. To highlight the apparent complexity of this process, consider the puzzle of Fig. 1.1. By choosing an initial run, and then progressively adding one run at a time, ensuring that each "new" run intersects with at least one currently present run, the number of possible arrangements of values that satisfy the total and non-duplication constraints can be found.

Table 3.4: The effect of number of runs present on number of solutions

Number of Runs Present	Number of Valid Value Arrangements
1	24
2	756
3	2,160
4	2,592
5	1,320
6	404
7	216
8	20
9	20
10	16
11	1
12	1

Table 3.4 shows this process specifically for the grid of Fig. 1.1, where the run ($k_{5,3}$,$k_{5,4}$,$k_{5,5}$,$k_{5,6}$) with run-total 10 was initially chosen. The number of potentially valid arrangements of val-

ues rises rapidly, before falling more gradually as the last runs and constraints are taken into account. This behaviour is more marked as the size of the puzzle grid increases, due to the increased number of runs and run intersections present. Puzzles with a higher level of difficulty may also contain more runs that can be satisfied using a higher number of ordered arrangements of values that meet their run-total requirement, meaning a higher number of potentially valid arrangements can be expected for "incomplete" grids. The number of runs that can be removed without affecting the uniqueness of a solution is highly puzzle specific; for this particular grid, a unique solution can be reached when the selected first 11 of the 12 runs are present. In general, changing the order in which runs are added may affect how quickly uniqueness is specified.

3.2 The Number of Valid Grid Arrangements

Depending on the constraint(s) currently enforced, a given grid may possess a number of valid "solutions"; this section aims to ascertain which type of constraint has most effect in terms of reducing the upper bound on the number of valid grid arrangements of values that exist and the effect of both constraints on the number of solutions of puzzle grids. The full enumeration of a particular (small sized) grid is also given in Sections 3.4 to 3.4.3. Firstly, a trivial upper bound, U_1, on the number of valid arrangements of values in a Kakuro puzzle grid is given:

Lemma 3.1. *Let w be the number of white cells within a Kakuro grid. Then the number of valid arrangements of values in a Kakuro grid is $U_1 \leq 9^w$.*

Proof. Each white cell can take any of the numerical values in the range $1, \ldots, 9$. Hence a puzzle with w white cells can accept a maximum of 9^w differing arrangements of values. □

Recall the sample puzzle grid of Fig. 1.1 (seen in Fig. 3.1 below with run-totals removed), with sixteen white cells. In this grid, the number of valid arrangements of values is upper bounded by $U_1 \leq 9^{16} \approx 1.853 \times 10^{15}$.

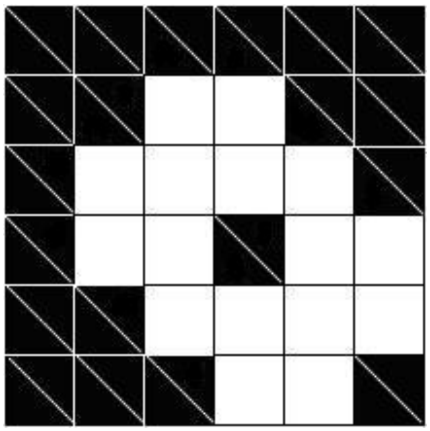

Figure 3.1: The sample puzzle grid of Fig. 1.1 with run-totals removed

3.2.1 Bounds Using Non-Duplication Constraints

The trivial upper bound, U_1 does not take into account the fact that cells within a run must contain distinct values. For an $n \times m$ grid of white cells, an improved upper bound, U_2, that depends on the highest possible number of values each cell can validly accept without violating the non-duplication constraint is found.

Lemma 3.2. *The number of valid ways of placing values from the range* $1, \ldots, 9$*, into an* $n \times m$ *grid of white cells* $(n \geq m)$ *is:*

$$U_2 \leq \begin{cases} \prod_{i=1}^{n} (9-i+1)^{(2i-1)} & \text{if } n = m \\ \dfrac{\prod_{i=1}^{n} (9-i+1)^{(2i-1)}}{\prod_{q=1}^{n-m} (9-q+1-m)^{(n+q-1)}} & \text{if } n > m \end{cases}$$

Proof. Cells within the same horizontal or vertical run must contain distinct values. However, two cells that are not within the same horizontal or vertical run, may accept the same value. If cells are considered from left to right and top to bottom, a cell at the intersection of a particular horizontal and vertical run is therefore restricted to contain only a value (from the standard range) that is not already present in the horizontal or vertical runs in which it resides. Consider the case in Fig. 3.2(a), cell entry $k_{3,3}$, for example, can accept a maximum of seven values, assuming equality of value v_1 in both the horizontal and vertical run (and

likewise for v_2), since the content of shaded cells have no affect on values placed in this cell. Consider Fig. 3.2(b), the number of values that each cell in this particular case may accept are shown, when all cells are considered in this way.

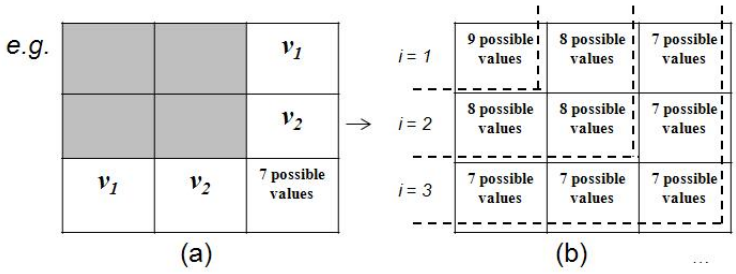

Figure 3.2: Finding the upper bound for a 3×3 example grid

Generalising this pattern, the first cell entry, $k_{1,1}$, in row $i = 1$, to be considered in a square grid may always contain any of the nine values from the standard range $1, \ldots, 9$. It can therefore accept $9 - i + 1 = 9$ values. Clearly, only this cell exists in a trivial 1×1 grid. In a 2×2 grid, the next largest square grid, the $2i - 1 = 3$ *new* cells (those that were not present in the smaller square grid), may accept $9 - i + 1 = 8$ values; they may not accept the one value already present in adjacent cells. These new cells appear in row 2 or column 2, so $i = 2$. In general, when all cells are considered in this way, the upper bound, U_2 is the product of the $9 - i + 1$ values that are placed in the $2i - 1$ new cells that are added each time the grid size increases. The grid size increases until $i = n$, the dimension of the overall grid itself. Hence for square grids, where $n = m$, the upper bound, U_2 on the number of valid arrangements of values that exists is the product of the number of values each cell can accept:

$$U_2 \leq \prod_{i=1}^{n} (9 - i + 1)^{(2i-1)} \tag{3.1}$$

Now consider $n > m$ (non-square grids), the upper bound, U_2, can be reduced since $2i-1$ *new* cells are *not* added when the grid size is increased from $(n-1) \times (n-1)$ to $n \times m$. Hence $n-m$ columns of n cells no longer need to be considered. These will be termed *additional* columns.

A cell at the top of each of $n-m$ additional columns can accept $9-q+1-m$ values for $q=1,\ldots,(n-m)$. Consider $q=n-m$, then there are $n+(q-1)$ cells in total within the additional columns that accept $9-q+1-m$ values; the rightmost column contains the same value n times, and it is also in the bottom cell of the previous $q-1$ additional columns. A similar analysis holds for all other values in other additional columns for $q=1,\ldots,(n-m)$. The upper bound based on the square grid, can therefore be divided by the product of the number of values that each of these cells from the additional columns could accept. Hence for non-square grids, where $n>m$, the upper bound, U_2, on the number of valid arrangements of values that exists is:

$$U_2 \leq \frac{\prod_{i=1}^{n}(9-i+1)^{(2i-1)}}{\prod_{q=1}^{n-m}(9-q+1-m)^{(n+q-1)}} \tag{3.2}$$

□

9 possible values	8 possible values	7 possible values	6 possible values
8 possible values	8 possible values	7 possible values	6 possible values
7 possible values	7 possible values	7 possible values	6 possible values
6 possible values	6 possible values	6 possible values	6 possible values

Figure 3.3: Finding the upper bound for a 4×3 example grid

For example, Fig. 3.3 shows a 4×3 grid. The inequality of 3.1 considers the grid as a square 4×4 grid. Therefore, $n-m=1$ columns of $n=4$ cells are unnecessarily considered. Therefore, the value that each of these cells contributes to the bound based on the square grid must be removed by dividing by the product of the number of values that may be placed into them.

Clearly if $m>n$, the grid may be transposed to satisfy $n\geq m$. Similarly, a lower bound, L_2, is now given that depends on the lowest possible number of values that a cell can validly

accept without contradicting the non-duplication constraint.

Lemma 3.3. *The number of valid ways of placing values from the range* $1, \ldots, 9$*, into an* $n \times m$ *grid of white cells is lower bounded by:*

$$L_2 \geq \prod_{x=9-n+1}^{9} \prod_{i=0}^{m-1} (x - i) \tag{3.3}$$

Proof. Cells within the same horizontal or vertical run must contain distinct values. However, two cells placed diagonally with respect to one another (so are not within the same horizontal or vertical run) may accept the same value. If cells are considered from left to right and top to bottom, a cell at the intersection of a particular horizontal and vertical run may therefore contain any value (from the standard range) that is not already present in the horizontal or vertical runs in which it resides. Consider the case as represented in Fig. 3.4(a). Assuming values v_1, v_2, v_3 and v_4 are distinct, cell entry $k_{3,3}$ can accept a maximum of five values, since the content of the shaded cells have no affect on the cell considered. Fig. 3.4(b) shows how many values each cell in this particular case may accept when all cells are considered in this way.

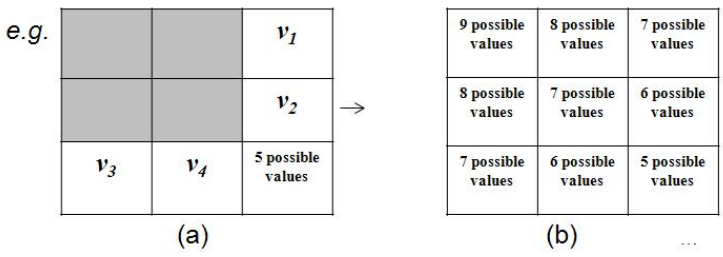

Figure 3.4: Finding the lower bound for a 3×3 example grid

The first cell entry, $k_{1,1}$, in row 1, to be considered in a square grid may always contain any of the nine values from the standard range $1, \ldots, 9$. Cells below this initial cell, can therefore accept a consecutively lower value until the value $9 - n + 1$ is reached in the bottom cell. Consider each row in turn and let the first element in the row be x. Each cell in the row can then accept a consecutively lower value than that placed in the left adjacent cell until

the cell in the rightmost column is reached, taking the value $(x-i)$ where $i=(m-1)$. The lower bound, L_2, on the number of valid arrangements of values that exists is the product of the number of values each cell can accept:

$$L_2 \geq \prod_{x=9-n+1}^{9} \prod_{i=0}^{m-1} (x-i) \tag{3.4}$$

□

The bounds of Lemma 3.2 and Lemma 3.3 apply only to $n \times m$ grids of white cells. Actual Kakuro puzzle grids do not typically consist of an $n \times m$ grid of white cells; they may contain black cells at the corners or sides of the grid, or may contain internal black cells. The puzzle grid structure can therefore vary widely between puzzles and so is highly puzzle specific. The current author notes that little progress seems to have been made in determining bounds for Survo puzzles [45] which possess a much simpler geometry (Survo puzzles are described in Section 2.2.4). However, similar upper and lower bound for an actual Kakuro puzzle grid, containing black cells, can be found. For the puzzle grid of Fig. 3.1, an upper bound similar to U_2 is $9^2 8^7 7^3 6^4 = 75,511,665,524,736$ and a lower bound similar to L_2 is $9^2 8^6 7^1 6^4 5^3 = 24,078,974,976,000$. The actual number of valid grids was found by the use of the counting program of Algorithm 3.1. This algorithm is exhaustive; it attempts to assign all possible values to all possible white cells, providing there are no duplicate values within horizontal and vertical runs. Each time a "successful" assignment is made, a counter is incremented. Using this counting program, the grid of Fig. 3.1 has 32,920,069,333,536 valid arrangements.

Algorithm 3.1: Valid Grid Arrangements

```
Initialise empty Cell_Stack and Counter = 0.
For Current_Value_Cell_1 from 1 to 9
  If placement of Current_Value_Cell_1 into Cell_Stack[1] will not cause duplication violations.
    Place Current_Value_Cell_1 into Cell_Stack[1].
    For Current_Value_Cell_2 from 1 to 9
      If placement of Current_Value_Cell_2 into Cell_Stack[2] will not cause duplication violations.
        Place Current_Value_Cell_2 into Cell_Stack[2].
...
  For Current_Value_Cell_w from 1 to 9
    If placement of Current_Value_Cell_w into Cell_Stack[w] will not cause duplication violations.
      Place Current_Value_Cell_w into Cell_Stack[w].
      Counter increased by one.
    EndIf
  EndFor
...
EndIf
EndFor
```

3.2.2 Bounds Using Run-Total Constraints

The upper bounds U_1 and U_2 of Section 3.2.1 do not take into account the fact that cells within a run must sum to a required run-total. In general, most cells cannot contain all values in the standard range $1, \ldots, 9$, particularly runs that sum to a very high or very low run-total. The bounds can therefore be greatly reduced by considering which of the nine available values can legitimately be placed in each of the (white) cell entries, $k_{i,j} \in W$, depending on both runs, r_{l_1} and r_{l_2}, in which the cell resides. l_1 and l_2 represent the run labels that identify a particular run, such that $1 \leq l_1 < l_2 \leq p$, where p is the number of runs within the puzzle. Let runs r_{l_1} and r_{l_2} have corresponding run-totals t_{l_1} and t_{l_2}

respectively. Two sets, $P_{i,j,1}$ and $P_{i,j,2}$, may be assigned to all (white) cell entries, $k_{i,j}$, that contain values that correspond to the maximum value that can be placed within the cell. Let the set $P_{i,j,1}$ correspond to the horizontal run and the set $P_{i,j,2}$ correspond to the vertical run. If a cell belongs to a run of length one, then the only element placed into the corresponding set is the run-total for the run. Otherwise, the sets $P_{i,j,a}$ $(a = 1, 2)$ contain all values from the range $1, \ldots, 9$ that are less than the run-total for the run to which they correspond. Note that the non-duplication constraint is currently ignored.

For example, in the grid of Fig. 1.1, the cell entry at the uppermost left of the grid, $k_{2,3}$, is a member of a horizontal run (r_{l_1}) with run-total $t_{l_1} = 5$ and another, vertical run (r_{l_2}) with run-total $t_{l_2} = 11$. Concentrating on the horizontal run-total, (r_{l_1}), only a value in the range $1, \ldots, 4$ can be placed in this cell. It seems reasonable to conjecture that in general, the following upper bound, U_3, is an improvement on U_1 and U_2 due to the assumed presence of runs with a very low run-total. It is unlikely that every cell could contain every value in the standard range $1, \ldots, 9$, particularly puzzles with grid dimensions more typical of those published. This would imply that all runs are associated to run-totals that are greater than nine. In such rare cases, $U_3 = U_1$ and $U_3 > U_2$. For example, consider the grid of Fig. 3.5 (shown with solution). The current upper bound $U_3 = U_1 = 9^7$ but the previous upper bound $U_2 = 9^1 8^3 7^3$.

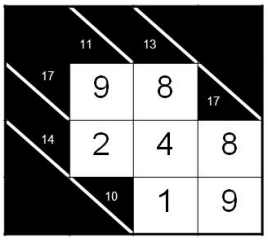

Figure 3.5: An example of when U_3 is not an improvement on U_1 and U_2

Lemma 3.4. *The number of valid arrangements of values in a Kakuro grid is:*

$$U_3 \leq \prod_{\forall k_{i,j} \in W} min\{|P_{i,j,a}| \quad | \, a = 1, 2\}$$

Proof. Assuming trivial runs of length one are considered to be valid runs, all cells are members of two runs, r_{l_a} (where $a = 1, 2$ and $1 \leq l_1 < l_2 \leq p$) with corresponding run-total

t_{l_a}, so two sets of values, $P_{i,j,a}$ may be assigned to all (white) cell entries $k_{i,j}$ ($\in W$). Let $P_{i,j,1}$ correspond to the horizontal run and $P_{i,j,2}$ correspond to the vertical run. Sets $P_{i,j,a}$ are assigned such that:

$$P_{i,j,a} = \{1, \ldots, 9\} \qquad t_{l_a} > 9$$

$$P_{i,j,a} = \{1, \ldots, t_{l_a} - 1\} \qquad t_{l_a} \leq 9, |r_{l_a}| \neq 1$$

$$P_{i,j,a} = \{t_{l_a}\} \qquad |r_{l_a}| = 1$$

The improved upper bound, U_3, then follows by finding and taking the product of the sizes of the smallest set belonging to each cell.

□

In the grid of Fig. 1.1, for example, $U_3 \leq 9^5 5^2 4^1 3^3 2^5 = 5,101,833,600$ arrangements can be calculated when all cells are considered in this way. Note that this is considerably less than U_1 and U_2 and is also considerably less than the actual number of valid arrangements of values that can be placed into the grid of Fig. 1.1 when *only* non-duplication constraints are satisfied (32,920,069,333,536, as calculated in Section 3.2.1).

Calculation of the current upper bound, U_3, assumes that a cell can contain any value from the standard range that is less than the lowest of the two run-totals associated to it. This is with the exception of cells that are one cell in length which can only accept one value. A better approach would be to rule out the values that, despite being lesser in value than the lowest run-total associated with the cell, still cannot be placed into the cell. For example, if a run of length two had an associated run-total of six, the value 3 could not be placed in either cell belonging to this run because it would result in a duplicated value. Each and every white cell within the puzzle grid belongs to two runs (one horizontal and one vertical), so a cell entry, $k_{i,j}$, can possibly contain values from two candidate sets; $C_{i,j,1}$, corresponding to the horizontal run and $C_{i,j,2}$, corresponding to the vertical run.

Lemma 3.5. *Let $C_{i,j,1}$ contain the values in the candidate set belonging to the horizontal run and $C_{i,j,2}$ the vertical run for cell entry $k_{i,j}$. The number of valid arrangements of values in a Kakuro grid is:*

$$U_4 \leq \prod_{\forall k_{i,j} \in W} | C_{i,j,1} \cap C_{i,j,2} |$$

Proof. Let $C_{i,j,1}$ contain the values in the candidate set belonging to the horizontal run and $C_{i,j,2}$ the vertical run, such that $C_{i,j,1}, C_{i,j,2} \subseteq \{1, \ldots, 9\}$ for each cell entry $k_{i,j}$. Therefore, the actual possibilities for the values that may be placed into a particular cell entry, $k_{i,j}$, are from a set obtained through the intersection of these two candidate sets: $C_{i,j,1} \cap C_{i,j,2}$. U_4 then follows by taking the product of the sizes of all such intersecting candidate sets. □

In the grid of Fig. 1.1, the cell entry $k_{2,3}$ is a member of a horizontal run (r_{l_1}) with run-total $t_{l_1} = 5$ and vertical run (r_{l_2}) with run-total $t_{l_2} = 11$. A run-total of five over two cells can be obtained by using an arrangement of $\{1, 4\}$ or an arrangement of $\{2, 3\}$, so cell entry $k_{2,3}$ can take any of the values in the candidate set $C_{2,3,1} = \{1, 2, 3, 4\}$ based on run r_{l_1}. Similarly, a run-total of eleven over four cells can only be obtained by using an arrangement of $\{1, 2, 3, 5\}$ so can accept any of the values in the candidate set $C_{2,3,2} = \{1, 2, 3, 5\}$ based on run r_{l_2}. Therefore, since $\{1, 2, 3, 4\} \cap \{1, 2, 3, 5\} = \{1, 2, 3\}$, only one of these three values can be added validly to this cell. Cell entry $k_{2,3}$ would therefore be assigned a "score" of three.

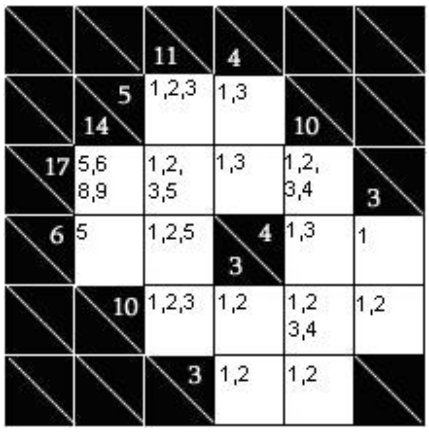

Figure 3.6: Run intersections for each white cell of Fig. 1.1

The grid of Fig 1.1 is shown again in Fig 3.6. For each cell, the elements present in the intersection of the two candidate sets are also shown. An improved upper bound $U_4 \leq 4^4 3^3 2^7 = 884,736$ arrangements can be calculated for this particular example. Clearly, the run-total constraint has most effect in terms of improving bounds. Candidate sets are used in Section 3.4 during the full enumeration of 2×2 puzzle grids and to inform pruning conditions that are incorporated into a backtracking solver for the automated solution of

Kakuro puzzles in Sections 5.5.2, 5.5.3 and 5.5.4.

3.2.3 The Diagonal Pairs Method

If the run-totals are removed from a puzzle grid, and hence the requirement of the values to sum to a given run-total is relaxed, an exact number of valid arrangements, one where only the non-duplication of values within a run is taken into consideration, can be found.

For most grids, the actual number of valid ways to place values into each and every white cell without causing a constraint violation (a duplicated value) cannot be simply enumerated. Since the non-duplication constraint states that the same numerical value cannot appear more than once in any horizontal or vertical run, equal values may be placed within cells that are situated diagonally from one another. The problem is therefore split into smaller problems, or cases since diagonal cells may or may not contain equal numerical values. Before considering larger Kakuro puzzle grids, the smallest problem of a 2×2 grid, and augmentations thereof, is considered in order to formulate a method of solution. Polynomials for the number of valid ways of placing values from the range $1, \ldots, x$ ($x \in \mathbb{Z}$) into the cells within each grid are determined.

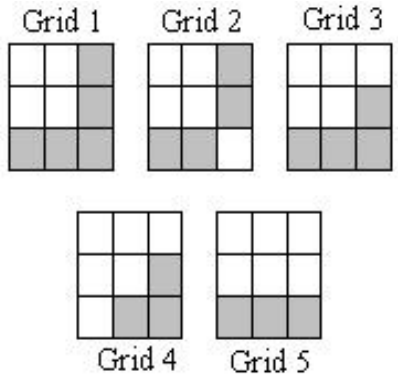

Figure 3.7: $K_{(2\times 2)}$ grid and augmentations

Let a *negative diagonal* pair of cells be any two cells that are not in the same row or the same column where the rightmost cell is placed at least one row above that of the leftmost cell. Similarly, let a *positive diagonal* pair of cells be any two cells that are not in the same

row or the same column where the rightmost cell is placed at least one row below that of the leftmost cell.

Lemma 3.6. *The number of possible arrangements for the placement of integer values from the range* $1, \ldots, x$ *in which there are no duplicate values in any horizontal run or any vertical run for:*

1. *A* $K_{(2\times 2)}$ *grid [Grid 1 in Fig. 3.7] is* $x(x-1)(x^2-3x+3)$,
2. *A* $K_{(2\times 2)}$ *grid augmented with cell entry* $k_{3,3}$ *[Grid 2 in Fig. 3.7] is* $x^2(x-1)(x^2-3x+3)$,
3. *A* $K_{(2\times 2)}$ *grid augmented with cell entry* $k_{1,3}$ *[Grid 3 in Fig. 3.7] is* $x(x-1)(x-2)(x^2-3x+3)$,
4. *A* $K_{(2\times 2)}$ *grid augmented with cell entries* $k_{1,3}$ *and* $k_{3,1}$ *[Grid 4 in Fig. 3.7] is* $x(x-1)(x-2)^2(x^2-3x+3)$,
5. *A* $K_{(2\times 3)}$ *grid [Grid 5 in Fig. 3.7] is* $x(x-1)(x-2)(x^3-6x^2+14x-13)$.

Proof. Since interest lies in the equality or non-equality of diagonal cells, it is possible to divide the problem into a number of different *cases* and *sub-cases*. Consider the equality or non-equality of a negative diagonal pair of cells (and combinations of two or more of such pairs) as representing distinct cases and let positive diagonal pairs (or combinations of two or more of such pairs) represent sub-cases which further divide the cases. Sub-cases are only taken into consideration when there are white cells in row three or below ($i \geq 3$). The grid may be thought of as an excerpt of a full puzzle grid.

The cases will be considered separately:

1. Elements within runs $(k_{1,1}, k_{1,2})$, $(k_{2,1}, k_{2,2})$, $(k_{1,1}, k_{2,1})$ and $(k_{2,1}, k_{2,2})$ cannot contain equal values. However, values placed in the pairs of cells $k_{1,1}$ and $k_{2,2}$, $k_{2,1}$ and $k_{1,2}$ may accept equal values since they are in distinct runs. Without loss of generality, consider the equality or non-equality of cell entries $k_{2,1}$ and $k_{1,2}$ (a negative diagonal pair of cells). The problem is split into two cases (sub-cases are not taken into consideration as there are no white cells in row three or below). In the first case, cell entry $k_{2,1}$ contains an equal value to cell $k_{1,2}$, therefore the number of possible arrangements of the values is $x(x-1)^2$. In the second case, cell entry $k_{2,1}$ does not contain an equal

value to cell entry $k_{1,2}$ so the number of possible arrangements is $x(x-1)(x-2)^2$. Combining the two cases of $K_{(2\times 2)}$, the total number of possible valid arrangements of values in $K_{(2\times 2)}$ is given by:

$$x(x-1)[(x-1)+(x-2)^2] = x(x-1)(x^2-3x+3) \tag{3.5}$$

2. Grid 2 is obtained by augmenting $K_{(2\times 2)}$ with a single cell entry, $k_{3,3}$, that is not connected to any horizontal run or vertical run in $K_{(2\times 2)}$. Therefore, since any of the x values could be placed into this new cell, the number of arrangements of $K_{(2\times 2)}$ (given in 3.5) is multiplied by x. Hence the total number of possible valid arrangements of grid 2 is given by:

$$x^2(x-1)(x^2-3x+3) \tag{3.6}$$

3. Grid 3 is obtained by augmenting $K_{(2\times 2)}$ with a cell entry, $k_{1,3}$, that is connected to one horizontal run, $(k_{1,1},\ k_{1,2})$. The value to be placed in $k_{1,3}$ must be distinct from the values placed in cell entries $k_{1,1}$ and $k_{1,2}$, which are themselves distinct. Therefore, since $(x-2)$ values could be placed into $k_{1,3}$, the number of arrangements of $K_{(2\times 2)}$ (given in 3.5) is multiplied by $(x-2)$. Hence the total number of possible valid arrangements is given by:

$$x(x-1)(x-2)(x^2-3x+3) \tag{3.7}$$

4. Grid 4 is obtained by augmenting $K_{(2\times 2)}$ with two cells; $k_{1,3}$ is now connected to horizontal run $(k_{1,1}, k_{1,2})$ of $K_{(2\times 2)}$ and $k_{3,1}$ is now connected to the vertical run $(k_{1,1}, k_{2,1})$. Any value placed in cell entry $k_{1,3}$ would depend on the distinct values placed in cells $k_{1,1}$ and $k_{1,2}$. Similarly, any value placed in cell $k_{3,1}$ would depend on the distinct values placed in cell entries $k_{1,1}$ and $k_{2,1}$. Therefore, since $(x-2)$ values could be placed in either of these new cells, the number of arrangements of $K_{(2\times 2)}$ (given in 3.5) is multiplied by $(x-2)^2$. Hence the total number of possible valid arrangements is given by:

$$x(x-1)(x-2)^2(x^2-3x+3) \tag{3.8}$$

5. Elements within runs $(k_{1,1}, k_{1,2}, k_{1,3})$, $(k_{2,1}, k_{2,2}, k_{2,3})$, $(k_{1,1}, k_{2,1})$, $(k_{1,2}, k_{2,2})$ and $(k_{1,3}, k_{2,3})$ cannot contain equal values. However, the same values are permitted in cells that are not horizontally or vertically adjacent to one another since they are in distinct runs. Without loss of generality, consider the equality or non-equality of

negative diagonal pair of cells $k_{1,2}$ and $k_{2,1}$ in combination with the equality or non-equality of negative diagonal pair $k_{1,3}$ and $k_{2,1}$ and pair $k_{1,3}$ and $k_{2,2}$. The problem is therefore split into five cases.

(a) Consider $k_{1,2} = k_{2,1}$, $k_{1,3} \neq k_{2,1}$ and $k_{1,3} \neq k_{2,2}$. The number of possible arrangements is:

$$x(x-1)(x-2)^2(x-3) \tag{3.9}$$

(b) Consider $k_{1,2} \neq k_{2,1}$, $k_{1,3} \neq k_{2,1}$ and $k_{1,3} = k_{2,2}$. The number of possible arrangements is:

$$x(x-1)(x-2)^2(x-3) \tag{3.10}$$

(c) Consider $k_{1,2} \neq k_{2,1}$, $k_{1,3} = k_{2,1}$ and $k_{1,3} \neq k_{2,2}$. The number of possible arrangements is:

$$x(x-1)(x-2)^3 \tag{3.11}$$

(d) Consider $k_{1,2} = k_{2,1}$, $k_{1,3} \neq k_{2,1}$ and $k_{1,3} = k_{2,2}$. The number of possible arrangements is:

$$x(x-1)(x-2)^2 \tag{3.12}$$

(e) Finally, consider $k_{1,2} \neq k_{2,1}$, $k_{1,3} \neq k_{2,1}$ and $k_{1,3} \neq k_{2,2}$. The number of possible arrangements is:

$$x(x-1)(x-2)(x-3)^3 \tag{3.13}$$

Combining the five cases (expressions 3.9 - 3.13) and factorising, the total number of possible valid arrangements of values within a $K_{(2\times3)}$ grid is given by:

$$x(x-1)(x-2)(x^3-6x^2+14x-13) \tag{3.14}$$

□

If values from the standard range $1, \ldots, 9$ were to be placed into the grids of Fig. 3.7, the number of valid arrangements of values, ensuring that there are no duplication violations are:

1. 4,104 for Grid 1
2. 36,936 for Grid 2
3. 28,728 for Grid 3

4. 201,096 for Grid 4

5. 179,424 for Grid 5

This approach is now used to determine how many ways there are to place values in a larger puzzle grid so that there are no constraint violations caused by the duplication of values (noting that run-totals are ignored). The example 3×3 puzzle grid, shown without run-totals in Fig. 3.8 is considered.

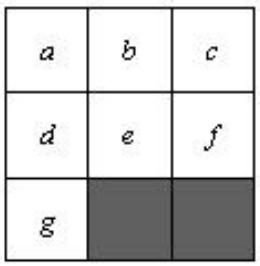

Figure 3.8: A 3×3 Kakuro grid with run-totals removed

Example 3.7. *The number of possible arrangements for the placement of integer values from the range* $1, \ldots, x$*, in which there are no duplicate values in any horizontal run or any vertical run, for the grid of Fig. 3.8 is given by* $x(x-1)(x-2)^2(x^3-6x^2+14x-13)$.

This can be seen by using the notation of Section 3.2.3. For ease, cells are labelled with and referred to by a unique letter label. The negative diagonal pairs are bd*,* bg*,* cd*,* ce*,* cg*,* eg *and* fg*. The positive diagonal pairs are* ae*,* af *and* bf*. Therefore, the cases are:*

- $b = d$,
- $b = g$,
- $c = d$,
- $c = e$,
- $c = g$,
- $e = g$,
- $f = g$,
- $b = d$ *and* $c = e$,
- $b = d$ *and* $c = e$,
- $b = d$ *and* $c = g$,

- $b = d$ *and* $e = g$,
- $b = d$ *and* $f = g$,
- $b = g$ *and* $c = d$,
- $b = g$ *and* $c = e$,
- $b = g$ *and* $f = g$,
- $c = d$ *and* $e = g$,
- $c = d$ *and* $f = g$,
- $c = e$ *and* $e = g$ *and* $c = g$,
- $b = d$ *and* $c = e$ *and* $f = g$,
- $c = e$ *and* $e = g$ *and* $c = g$ *and* $b = d$,
- $b = g$ *and* $c = e$ *and* $f = g$,
- $b = g$ *and* $c = d$ *and* $f = g$,
- *none of the negative diagonal pairs are equal.*

Each case has five sub-cases, based on the positive diagonal pairs.

- $a = e$,
- $a = f$,
- $b = f$,
- $a = e$ *and* $b = f$,
- *none of the positive diagonal pairs are equal.*

Working left-to-right and top-to bottom, using the enumeration technique of Lemma 3.6, the number of possible arrangements of values for each case is given:

- *Case 1:* $2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- *Case 2:* $2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- *Case 3:* $3x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)$,
- *Case 4:* $2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- *Case 5:* $3x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)$,
- *Case 6:* $2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- *Case 7:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- *Case 8:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- *Case 9:* $x(x-1)(x-2)(x-3)(x-4) + 2x(x-1)(x-2)(x-3)$,
- *Case 10:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- *Case 11:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- *Case 12:* $x(x-1)(x-2)(x-3)(x-4) + 2x(x-1)(x-2)(x-3)$,
- *Case 13:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,

- *Case 14:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- *Case 15:* $x(x-1)(x-2)(x-3)(x-4) + 2x(x-1)(x-2)(x-3)$,
- *Case 16:* $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- *Case 17:* $x(x-1)(x-2)(x-3)(x-4)$,
- *Case 18:* $x(x-1)(x-2)(x-3)(x-4) + 2x(x-1)(x-2)(x-3)$,
- *Case 19:* $x(x-1)(x-2)(x-3)$,
- *Case 20:* $x(x-1)(x-2)(x-3) + x(x-1)(x-2)$,
- *Case 21:* $x(x-1)(x-2)(x-3)$,
- *Case 22:* $x(x-1)(x-2)(x-3)$,
- *Case 23:* $x(x-1)(x-2)(x-3)(x-4)[(x-5)(x-6) + 3(x-5) + 1]$.

Hence, following simplification, there are $x(x-1)(x-2)^2(x^3 - 6x^2 + 14x - 13)$ *valid ways of placing values without duplication.*

If the usual range of values, $1, \ldots, 9$ *is used, this means there are* $1,255,968$ *valid arrangement of values.*

3.2.4 Adapting the Diagonal Pairs Approach for Larger Grids

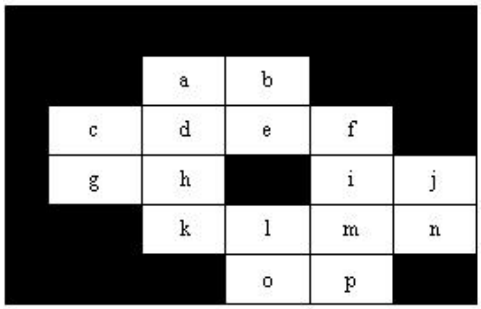

Figure 3.9: The puzzle grid of Fig. 1.1 with run-totals removed

For puzzles with small grid sizes, typically those with grid dimension 3×3 or smaller, the above diagonal pairs method of Section 3.2.3 can be used to count how many ways exist to place values from the range $1, \ldots, x$ ($x \in \mathbb{Z}$) into each cell without duplication in runs. This approach is highly inefficient and cumbersome for larger puzzle grids due to the number of negative and positive diagonal pairs, and hence the high number of cases and sub-cases

present. The method was implemented for the grid of Fig. 3.9. There were 1,308 cases found before the method was discontinued, even though no positive diagonals and only single, double and some triple combinations of negative diagonals had been examined at this point. To demonstrate the cumbersome nature of this infeasible method, Appendix A lists the cases derived for the puzzle of Fig. 3.9 before discontinuation.

As an alternative, a puzzle grid may be split into smaller disjoint sub-grids, where a sub-grid is small enough for the Diagonal Pairs method to be effectively used. Before considering the puzzle grid of Fig. 3.9, a smaller grid is considered.

3.2.4.1 Initial Findings for a Smaller Grid

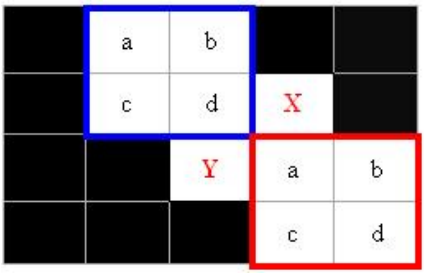

Figure 3.10: Disjoint sub-grids of a smaller sample puzzle grid

Consider a large grid to be split into disjoint smaller sub-grids, where certain cells may be temporarily ignored to ensure disjointness. A smaller grid, (shown in Fig. 3.10), was firstly considered where one or both "ignored" cells (X and Y) are progressively re-added to the overall grid. Each disjoint grid is a 2×2 grid, previously considered in Section 3.2.3. The number of valid arrangements of the values $1, \ldots, x$ ($x \in \mathbb{Z}$) within such a grid is given by the formula $x(x-1)(x^2-3x+3)$ shown in Lemma 3.6. The disjoint sub-grids are identical. Since x, the highest available value in the valid range, is the only variable present, the number of valid arrangements of values in the range $1, \ldots, x$ that can be added to the grid of Fig. 3.10 (with or without cells X and Y present), denoted by P, is represented by an equation in the form:

$$P = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6 + hx^7 ... \tag{3.15}$$

where $a, b, \ldots$ are constants to be determined and x denotes the highest available value to be placed in the cells. The order of a specific equation is equal to the number of white cells present.

Lemma 3.8. *Consider the grid of Fig. 3.10 without cells X and Y. The number of possible arrangements for the placement of integer values from the range $1, \ldots, x$, in which there are no duplicate values in any horizontal run or any vertical run is given by $[x(x-1)(x^2-3x+3)]^2$.*

Proof. Using the diagonal pairs method, the number of valid arrangements of values within each 2×2 disjoint sub-grid was shown in Lemma 3.6 to be $x(x-1)(x^2-3x+3)$. A grid comprising of two such disjoint grids will have $[x(x-1)(x^2-3x+3)]^2$ arrangements, since for any value arrangement within the upper sub-grid, $x(x-1)(x^2-3x+3)$ arrangements exist within the lower sub-grid. □

Cell X will now be considered while cell Y remains ignored.

Lemma 3.9. *Consider the Grid of Fig. 3.10 with cell Y ignored. The number of possible arrangements for the placement of integer values from the range $1, \ldots, x$, in which there are no duplicate values in any horizontal run or any vertical run is given by $x[(x-1)(x-2)(x^2-3x+3)]^2$.*

Proof. Consider the grid of Fig. 3.10 with cell Y ignored. Cells within the bottom row of the upper disjoint sub-grid are limited in respect to what values they can accept; they can not accept the value placed in cell X. The diagonal pairs method (see Appendix B) gives the number of valid arrangements for each *constrained* sub-grid as $(x-1)(x-2)(x^2-3x+3)$. The linear terms of the unconstrained disjoint sub-grid, those that correspond to these two constrained cells, are decreased by one to obtain the linear terms in the constrained sub-grid. The lower disjoint sub-grid has an identical number of arrangements since it shares rotational symmetry with the upper grid. Since for any value arrangement within the upper sub-grid, $(x-1)(x-2)(x^2-3x+3)$ arrangements exist within the lower sub-grid and since cell X can take any of x values, the overall total number of arrangements is $x[(x-1)(x-2)(x^2-3x+3)]^2$. □

This result has been computationally confirmed using Algorithm 3.1 of Section 3.2.1 for a number of alternative value ranges.

The number of arrangements of values for the overall grid of Fig. 3.10 cannot be derived from inspection or easily by re-examination of cases and sub-cases. As an alternative, Gaussian elimination is used to solve five equations of the form of Equation 3.15 for each disjoint, constrained sub-grid. Certain cells can no longer accept the value placed in cell X, cell Y or both. The exhaustive counting program (Algorithm 3.1 of Section 3.2.1) is used to determine how many valid arrangements exist for the placement of values from a number of alternative ranges into one of the sub-grids, when X and Y contain either equal or distinct values. Since each sub-grid contains four cells, the corresponding equation of the form of Equation 3.15, showing the number of such valid arrangements, is of order 4 so requires only terms with coefficients $a, \ldots, e$. The *msolve* function, which solves equations given in matrix form, within the mathematical package *Maple 12* [40] is used to calculate $a, \ldots, e$ in the equation, since x (the highest value in the range of values used) is known. Therefore, the roles of coefficients and variables are reversed; "x^0", ..., "x^4" now represent coefficients of variables $a, \ldots, e$.

Conjecture 3.10. *Consider the complete grid of Fig. 3.10. The number of possible arrangements for the placement of integer values from the range* $1, \ldots, x$*, in which there are no duplicate values in any horizontal run or any vertical run, is given by:*

$$x(x-1)(x-2)^2(x^6 - 11x^5 + 53x^4 - 144x^3 + 235x^2 - 221x + 96) \tag{3.16}$$

Evidence of this conjecture follows. Firstly, let cells X and Y contain distinct values.

$$\begin{bmatrix} 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \\ 1 & 6 & 36 & 216 & 1296 \\ 1 & 7 & 49 & 343 & 2401 \\ 1 & 8 & 64 & 512 & 4096 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 18 \\ 87 \\ 268 \\ 645 \\ 1326 \end{bmatrix}$$

The above represents a system of equations $XA = B$. Recall that values within B represent the number of valid arrangements of values, using a number of alternative ranges, that exist for placement into the grid of Fig. 3.10, and have been found using the exhaustive counting program (Algorithm 3.1) of Section 3.2.1. For example, the first equation corresponds to

$a + 4b + 16c + 64d + 256e = 18$; there are 18 ways of placing the values $1, \ldots, 4$ into one of the sub-grids. Likewise, there are 87 ways of placing the values $1, \ldots, 5$ into one of the sub-grids *etc.*

Let R be an expression for the number of valid arrangements of values within the grid of Fig. 3.10 where cells X and Y contain distinct values. Substituting values obtained, using Maple, for variables a through e into an equation of the form 3.15 gives:

$$R = 22 - 37x + 25x^2 - 8x^3 + x^4 \tag{3.17}$$

Similarly, when cells X and Y contain the the same value.

$$\begin{bmatrix} 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \\ 1 & 6 & 36 & 216 & 1296 \\ 1 & 7 & 49 & 343 & 2401 \\ 1 & 8 & 64 & 512 & 4096 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 30 \\ 120 \\ 340 \\ 780 \\ 1554 \end{bmatrix}$$

Let S be an expression for the number of valid arrangements of values within the grid of Fig. 3.10 where cells X and Y contain an equal value. Substituting values obtained for variables a through e into an equation of the form 3.15 gives:

$$S = 10 - 23x + 19x^2 - 7x^3 + x^4 \tag{3.18}$$

If X and Y contain equal values, then they may contain x values. If X and Y contain distinct values, $x(x-1)$ combinations exist. The total number of arrangements, using equations 3.17 and 3.18, for the whole of the grid of Fig. 3.10 is therefore:

$$x(x-1)R^2 + xS^2 \tag{3.19}$$

which, following simplification, is:

$$x(x-1)(x-2)^2(x^6 - 11x^5 + 53x^4 - 144x^3 + 235x^2 - 221x + 96) \tag{3.20}$$

This equation is of order 10, which is equal to the number of cells within the grid of Fig. 3.10.

If $x = 9$, that is the standard range of values $1, \ldots, 9$ were to be used, this grid would have 500,273,928 valid arrangements of values. This result is verified using the exhaustive counting program (Algorithm 3.1) of Section 3.2.3.

3.2.4.2 Enumerating the Grid of Fig. 3.9

Using knowledge obtained from the analysis of the smaller grid of Section 3.2.4.1, the grid of Fig. 3.9 is split into identical upper and lower disjoint sub-grids (shown in Fig. 3.11).

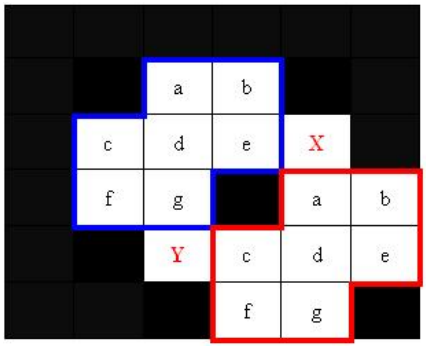

Figure 3.11: Disjoint sub-grids of Fig. 3.9

Each disjoint sub-grid of Fig. 3.11 has $x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)$ valid arrangements of values from the range $1,\ldots,x$ (shown in Appendix C).

Lemma 3.11. *Consider the grid of Fig. 3.11 without cells X and Y. The number of possible arrangements for the placement of integer values from the range $1,\ldots,x$, in which there are no duplicate values in any horizontal run or any vertical run is given by $[x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)]^2$.*

Proof. Since each sub-grid of Fig. 3.11 has $x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)$ valid arrangements and since sub-grids are disjoint, there will be $[x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)]^2$ valid arrangements for the grid of Fig. 3.11 without cells X and Y, since for any of the $x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)$ arrangements of values within the upper sub-grid, $x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)$ arrangements exist within the lower sub-grid.

□

Lemma 3.12. *Consider the grid of Fig. 3.11 with cell Y ignored. The number of possible arrangements for the placement of integer values from the range $1,\ldots,x$, in which there are no duplicate values in any horizontal run or any vertical run is given by $x[(x-1)(x-2)(x-3)(x^4-7x^3+20x^2-28x+17)]^2$.*

Proof. Consider the grid of Fig. 3.11 with cell Y ignored. Cells within the middle row of the upper disjoint sub-grid are limited in respect to what values they can accept; they can not accept the value placed in cell X. The diagonal pairs method (see Appendix D), gives the number of arrangements for each *constrained* sub-grid of Fig. 3.11 as $(x-1)(x-2)(x-3)(x^4-7x^3+20x^2-28x+17)$. The linear terms of the unconstrained, disjoint sub-grid, those that correspond to the three constrained cells, are decreased by one to obtain the linear terms in the constrained sub-grid. The lower sub-grid shares rotational symmetry so possesses an identical number of arrangements. Since for any value arrangement within the upper sub-grid, $(x-1)(x-2)(x-3)(x^4-7x^3+20x^2-28x+17)$ arrangements exist within the lower sub-grid and since cell X can accept any of x values, the overall total number of arrangements is $x[(x-1)(x-2)(x-3)(x^4-7x^3+20x^2-28x+17)]^2$. □

This result is verified using the exhaustive counting program (Algorithm 3.1) of Section 3.2.1 for a number of alternative ranges of values.

The number of arrangements that exist for the overall grid of Fig. 3.11 cannot be derived from inspection or easily by re-examination of cases and sub-cases. As an alternative, Gaussian elimination is used to solve eight equations of the form of Equation 3.15 for each disjoint, constrained sub-grid. The exhaustive counting program (Algorithm 3.1 of Section 3.2.1) is used to determine how many valid arrangements exist for the placement of values from a number of alternative ranges into one of the sub-grids, when X and Y contain either equal or distinct values. Since each sub-grid contains seven cells, the corresponding equation of the form of Equation 3.15, showing the number of such valid arrangements, will be of order 7 so requires only terms with coefficients $a, \ldots, h$. The *msolve* function, within the mathematical package *Maple 12* [40] is again used to calculate $a, \ldots, h$ in the equation.

Conjecture 3.13. *Consider the complete grid of Fig. 3.11. The number of possible arrangements for the placement of integer values from the range* $1, \ldots, x$ $(x \in \mathbb{Z})$*, in which there are no duplicate values in any horizontal run or any vertical run, is given by:*

$$\begin{aligned} x(x-1)(x-2)^2(x-3)^2(x^{10}-21x^9+202x^8-1177x^7+4621x^6 \\ -12832x^5+25644x^4-36594x^3+35865x^2-21915x+6375) \end{aligned} \tag{3.21}$$

Evidence of this conjecture follows. Firstly, let cells X and Y contain distinct values.

$$\begin{bmatrix} 1 & 2 & 4 & 8 & 16 & 32 & 64 & 128 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 & 2187 \\ 1 & 4 & 16 & 64 & 256 & 1024 & 4096 & 16384 \\ 1 & 5 & 25 & 125 & 625 & 3125 & 15625 & 78125 \\ 1 & 6 & 36 & 216 & 1296 & 7776 & 46656 & 279936 \\ 1 & 7 & 49 & 343 & 2401 & 16807 & 117647 & 823543 \\ 1 & 8 & 64 & 512 & 4096 & 32768 & 262144 & 2097152 \\ 1 & 9 & 81 & 729 & 6561 & 59049 & 531441 & 4782969 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 40 \\ 1122 \\ 10104 \\ 53260 \\ 203520 \\ 626430 \end{bmatrix}$$

The above represents a system of equations $XA = B$. Recall that values within B represent the number of valid arrangements of values, using a number of alternative ranges, that exist for placement into the grid of Fig. 3.11, and have been found using the exhaustive counting program (Algorithm 3.1) of Section 3.2.1. For example, the first equation corresponds to $a + 2b + 4c + 8d + 16e + 32f + 64g + 128h = 0$; there are 0 valid ways of placing the values 1 and 2 into one of the sub-grids. Likewise, there are 0 ways of placing the values $1, \ldots, 3$ into one of the sub-grids, 40 ways of placing the values $1, \ldots, 4$ into one of the sub-grids *etc.*

Let R be an expression for the number of valid arrangements of values within the grid of Fig. 3.11 where cells X and Y contain distinct values. Substituting values obtained, using Maple, for variables a through h into 3.15 gives:

$$R = -528 + 1370x - 1583x^2 + 1055x^3 - 436x^4 + 111x^5 - 16x^6 + x^7 \tag{3.22}$$

Similarly, when cells X and Y contain the the same value.

$$\begin{bmatrix} 1 & 2 & 4 & 8 & 16 & 32 & 64 & 128 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 & 2187 \\ 1 & 4 & 16 & 64 & 256 & 1024 & 4096 & 16384 \\ 1 & 5 & 25 & 125 & 625 & 3125 & 15625 & 78125 \\ 1 & 6 & 36 & 216 & 1296 & 7776 & 46656 & 279936 \\ 1 & 7 & 49 & 343 & 2401 & 16807 & 117647 & 823543 \\ 1 & 8 & 64 & 512 & 4096 & 32768 & 262144 & 2097152 \\ 1 & 9 & 81 & 729 & 6561 & 59049 & 531441 & 4782969 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 78 \\ 1608 \\ 13020 \\ 64920 \\ 239610 \\ 720048 \end{bmatrix}$$

Let S be an expression for the number of valid arrangements of values within the grid of Fig. 3.11 where cells X and Y contain equal values. Substituting values obtained, using

Maple, for variables a through h into an equation of the form of 3.15 gives:

$$S = -222 + 731x - 1008x^2 + 767x^3 - 351x^4 + 97x^5 - 15x^6 + x^7 \tag{3.23}$$

If X and Y contain equal values, then they may contain x values. If X and Y contain distinct values, $x(x-1)$ combinations exist. The total number of valid arrangements of values, using equations 3.22 and 3.23, for the whole of the grid of Fig. 3.11 is therefore:

$$x(x-1)R^2 + xS^2 \tag{3.24}$$

which, following simplification, is:

$$\begin{aligned} x(x-1)(x-2)^2(x-3)^2(x^{10} - 21x^9 + 202x^8 - 1177x^7 + 4621x^6 \\ -12832x^5 + 25644x^4 - 36594x^3 + 35865x^2 - 21915x + 6375) \end{aligned} \tag{3.25}$$

This equation is of order 16, which is equal to the number of cells within the grid of Fig. 3.11.

This result is verified using the exhaustive counting program of Section 3.2.3. Using Equation 3.25, the grid of Fig. 3.11 can be filled in $32,920,069,333,536$ valid ways if values in the range $1, \ldots, 9$ are used.

3.3 A Generating Function for the Number of Unordered Arrangements of Values Within Runs

The positioning of runs, and the selections of run-totals of Kakuro puzzles can vary greatly between alternative puzzle grids. Therefore, the task of devising a general formula for the exact number of possible Kakuro grid arrangements of a given size is not considered.

Instead, attention is focused here on determining the total number of arrangements of values within a single run which would satisfy the puzzle constraints – the run-total constraint, and the requirement to have no duplicated values in the run. Firstly, consider Kakuro puzzles that use a range of values $1, \ldots, x$ that is beyond the usual range $1, \ldots, 9$.

Lemma 3.14. *The total number of valid, unordered arrangements of* $\mid r_l \mid$ *distinct values* ($\mid r_l \mid \leq x$) *for a specific run,* r_l*, with run-total* t_l*, is given by the coefficients of* $a^{|r_l|}y^{t_l}$*,*

obtained from the series expansion of the generating function:

$$\prod_{i=1}^{x}(1+ay^i)$$

Proof. The number of distinct, unordered arrangements of values for all run-lengths $\leq x$, that have the run-total t_l, is equivalent to that of the number of distinct integer partitions of t_l, given by the coefficients of y^{t_l} in the series expansion of the generating function given in [25]:

$$\prod_{i=1}^{x}(1+y^i)$$

This generating function does not currently show the number of ways a run-total, t_l, can be partitioned into a specific number of distinct parts, or a specific run-length.

Since each block of the partition may be described using the dummy variable a, the number of occurrences of a (the power of a) describes the number of blocks in the partition and hence the number of cells present in the run. Therefore, the coefficients of $a^{|r_l|}y^{t_l}$ describes the number of partitions of t_l into $\mid r_l \mid$ parts or the number of arrangements of $\mid r_l \mid$ distinct values in a run of length $\mid r_l \mid$ with run-total t_l. □

This generating function has been used to develop a look-up table that is employed in a heuristic in Section 5.2.1.

The runs belonging to standard Kakuro puzzles can be no greater than nine cells in length, since the range of values $1, \ldots, 9$ is used.

Lemma 3.15. *The total number of valid, unordered arrangements of* $\mid r_l \mid$ *distinct values* ($\mid r_l \mid \leq 9$) *for a specific run,* r_l*, with run-total* t_l*, is given by the coefficients of* $a^{|r_l|}y^{t_l}$ *obtained from the series expansion of the generating function:*

$$\prod_{i=1}^{9}(1+ay^i)$$

Proof. Follows directly from Lemma 3.14. □

Lemma 3.16. *Let* q *be the number of valid, unordered arrangements of* $\mid r_l \mid$ *distinct values* ($\mid r_l \mid \leq x$) *for a specific run* r_l *with run-total* t_l*, then the number of ordered arrangements of these values is given by:*

$$q \mid r_l \mid !$$

Proof. There are $\mid r_l \mid !$ ways of ordering $\mid r_l \mid$ distinct values. □

3.4 Using Run-Total and Non-Duplication Constraints

Each run within a puzzle can be assigned a candidate set; a set containing all values that can be used to satisfy a given run-total over the given number of cells. The production of a heuristic, based on the use of such candidate sets, that has been incorporated into an automated solver is given in Section 5.5.3. Candidate sets are now used to enumerate all 2×2 puzzle grids.

3.4.1 2×2 Grids with a Unique Solution

The candidate sets for all runs with given totals over two cells are shown below:

Run-total	Candidate Set
1:	[]
2:	[]
3:	[1, 2]
4:	[1, 3]
5:	[1, 2, 3, 4]
6:	[1, 2, 4, 5]
7:	[1, 2, 3, 4, 5, 6]
8:	[1, 2, 3, 5, 6, 7]
9:	[1, 2, 3, 4, 5, 6, 7, 8]
10:	[1, 2, 3, 4, 6, 7, 8, 9]
11:	[2, 3, 4, 5, 6, 7, 8, 9]
12:	[3, 6, 5, 7, 8, 9]
13:	[4, 5, 6, 7, 8, 9]
14:	[5, 6, 8, 9]
15:	[6, 7, 8, 9]
16:	[7, 9]
17:	[8, 9]

A "unique pairing" is a pair of runs (each of a certain, fixed length with an associated run-total) with the property that their candidate sets contain only one value in common. A

cell at the intersection of such a pair of runs can accept only one value. Let an *essentially different* grid be an arrangement of values within a grid that cannot be obtained by one or more reflections or rotations of a grid that has already been considered. In this section, the number of essentially different 2×2 grid arrangements is investigated. Without loss of generality, consider the intersection of the unique pairings to occur in cell entry $k_{1,2}$.

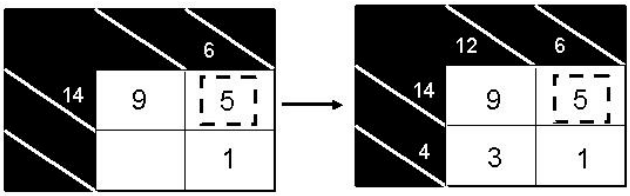

Figure 3.12: Using unique pairings within a 2×2 puzzle grid

For example, in Fig. 3.12, the two cells with run-total of fourteen can be filled using the tuples (5,9), (6,8), (8,6) and (9,5), meaning the corresponding candidate set is {5,6,8,9} for this run. Similarly, the intersecting two-cell run with run-total six can be filled using a value from the candidate set {1,2,4,5}. The intersection of these two candidate sets contains only a unique element, 5. Only this value can therefore be placed in cell entry $k_{1,2}$, the intersecting cell of the two runs. The placement of the values in cell entries $k_{1,1}$ and $k_{2,2}$ is then trivial; a 9 in cell $k_{1,1}$ and a 1 in $k_{2,2}$.

The remaining cell entry, $k_{2,1}$, can contain any of the seven values in the candidate set {2,3,4,5,6,7,8}, namely any value which does not equal values already placed in adjacent cells. Hence, using this "unique pairing" of runs with totals fourteen and six, there are seven possible essentially different valid puzzle grids that can be formed when the contents of the cell entry $k_{1,2}$ is fixed by such a unique pairing. Only following the placement of the final value (a 3 in Fig. 3.12 above) can the remaining two run-totals be added. It should be noted that the cell entry $k_{2,1}$ can accept a maximum of seven values and not eight. If an eighth value were possible, the implication would be that the values within the cell entries $k_{1,1}$ and $k_{2,2}$ are equal, further implying that the run-totals within the supposed unique pairing are equal; there is no unique pairing between two runs with equal run-total.

For two runs of length two (run a and run b say), there are thirteen unique pairings, shown

in Table 3.5. The unique element obtained by the intersection of the respective candidate sets in each case are also shown:

Table 3.5: Unique pairings for run pairs of length two, and corresponding unique element

a	b	unique element
3	4	[1]
4	6	[1]
3	11	[2]
4	11	[3]
4	12	[3]
5	13	[4]
6	14	[5]
7	15	[6]
8	16	[7]
9	16	[7]
9	17	[8]
14	16	[9]
16	17	[9]

For each of these thirteen unique pairings, the final cell to be considered in a 2×2 grid can accept seven values, as in the example of Fig. 3.12. This suggests that there are 13*7 = 91 essentially different valid arrangements of values for a 2×2 grid that possesses a unique solution. Recall that to avoid the multiple counting of grids that are rotations or reflections of others, unique pairings always govern the value to be placed in $k_{1,2}$. However, it is now shown that 91 is actually an upper bound for the number of essentially different puzzle grids with a unique solution that can be derived using these unique pairings.

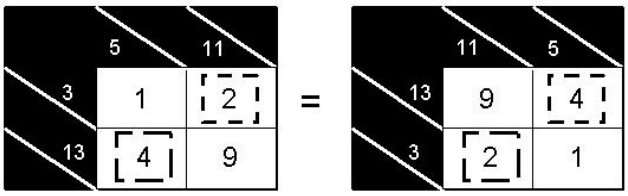

Figure 3.13: A grid that has been counted more than once

Consider Fig. 3.13, this demonstrates a case where an arrangement of values placed in a grid has been included more than once in the total of 91. The right-hand grid is obtained

from the left-hand grid following a 180° rotation. The left-hand grid is filled, as before, by placing a 2 in $k_{1,2}$ since runs with totals three and eleven form a unique pairing where 2 is the only element present in both candidate sets. The placement of the values in $k_{1,1}$ and $k_{2,2}$ is again trivial. The remaining cell can accept seven values; namely any value except a 1 and a 9. From these seven values, a 4 has been chosen in the figure.

The alternative, right-hand grid is now considered and is filled by placing a 4 in $k_{1,2}$, since runs with totals five and thirteen form a unique pairing where 4 is the only element present in both candidate sets. Correct values are again trivially placed in $k_{1,1}$ and $k_{2,2}$. In theory, seven values can be placed in the remaining cell, $k_{2,1}$, although as Fig. 3.13 shows, the placement of a 2 in this cell would result in the left-hand grid that has already been included in the total of 91.

There are 29 similar cases, meaning that 29 grids arrangements are included more than once in the total of 91. Therefore, there are 62 essentially different 2×2 puzzle grids, each with a unique solution that can be derived from the thirteen unique pairings.

3.4.2 Counting Further Grids

In Section 3.4, the thirteen "unique pairings" were used to find 62 essentially different puzzle grids that possessed a unique solution (so reflections and rotations of any one of these 62 grids did not count as new, distinct arrangements). Further pairs of runs are now investigated where the intersection of their corresponding candidate sets contain more than one value (up to a maximum size of eight). This introduces grid arrangements that possess the same horizontal and vertical run-totals, hence representing puzzles with multiple solutions. These can still be termed "puzzle grids" but are no longer *well-formed*. Without loss of generality, consider the intersection of the unique pairings to occur in cell entry $k_{1,2}$.

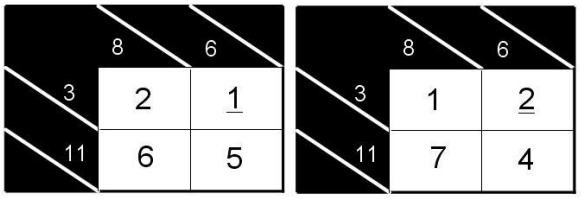

Figure 3.14: A grid possessing two solutions

Fig. 3.14 shows a grid with two solutions. The upper, right cell entry, $k_{1,2}$, is at the intersection of two runs with totals three and six, that have two elements in common in their respective candidate sets, namely 1 and 2. As before, if the first possibility, 1, is placed in $k_{1,2}$ (in the left-hand grid), correct values can again be trivially placed in cell $k_{1,1}$ and $k_{2,2}$. Any of seven values may be chosen for placement into the remaining cell entry $k_{2,1}$ (6 was chosen in the figure). If the process was repeated with the 2 (the other value present in the intersection of the candidate sets of runs of length two with totals three and six) placed in $k_{1,2}$, the choice of the value 7 in the remaining cell would result in two grids that have the same run-totals. Hence the grid possesses two valid solutions. Hence in this example, there were two values present in the intersection of the candidate sets corresponding to runs with totals three and six (where cell entry $k_{1,2}$ is the cell of intersection), so two grid arrangements, both a solution to the same puzzle grid are formed.

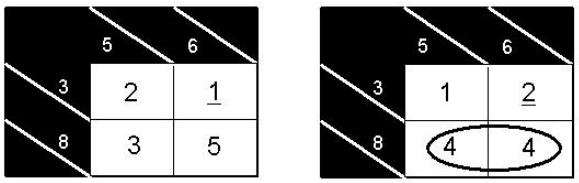

Figure 3.15: A new grid with a unique solution

Occasionally, as shown in Fig. 3.15, the number of grid arrangements obtained as solutions to a puzzle grid does not match the number of values in the intersection of the candidate sets belonging to the two runs that intersect at cell $k_{1,2}$. If the intersection of the candidate sets contain two values for example, a puzzle with two solutions is expected. However, as Fig. 3.15 shows, sometimes one of the two grids is invalid, resulting in a new grid arrangement with, in this case, one, unique solution (that should therefore be added to the 62 such

puzzles already considered). This situation also arises when there are more values in the intersection of the candidate sets belonging to cell entry $k_{1,2}$. For example, although there are five elements following the intersection of some candidate sets that belong to runs with specific run-totals, new puzzle grids that possess only four solutions can be derived. Table 3.7 shows how many grid arrangements were counted, each belonging to puzzle grids with varying numbers of valid solutions, following investigation of cases based on the size of the intersection of candidate sets. Table 3.6 shows the number of essentially different 2×2 grid arrangements formed, where the size of the intersection of the two candidate sets, between 1 and 8, is also given. The table also shows how many solutions the puzzle grid possesses. There are 666 essentially different 2×2 grids but only 79 of these possess a unique solution.

Table 3.6: The number of grids counted

Number of Solutions	1	2	3	4	5	6	7	8
1 intersection	62							
2 intersections	15	172						
3 intersections	2	39	111					
4 intersections	0	0	6	74				
5 intersections	0	0	0	36	65			
6 intersections	0	0	0	0	0	48		
7 intersections	0	0	0	0	0	3	21	
8 intersections	0	0	0	0	0	0	0	12
Total	79	211	117	110	65	51	21	12

3.4.3 Fully Enumerating all Valid, 2×2 Grid Arrangements

If rotations and reflections are to be considered as separate grids, the essentially different puzzle grids, derived in Section 3.4.2, would have to be counted either two, four or eight times, depending on the equality of values placed in one or both pairs of diagonal cells. The diagonal pairs of cell entries are $k_{1,1}$ and $k_{2,2}$, $k_{1,2}$ and $k_{2,1}$. The cases are shown below, where A, B, C and D represent distinct values:

•Neither of the values placed in the diagonal pairs are equal to one another

There are eight grids that can be formed by the rotational and reflective symmetry of the original grid:

A	B
C	D

C	A
D	B

D	C
B	A

B	D
A	C

C	D
A	B

B	A
D	C

A	C
B	D

D	B
C	A

•Values within only one of the diagonal pairs are equal to one another

There are four grids that can be formed by the rotational and reflective symmetry of the original grid:

A	B
B	C

C	B
B	A

B	C
A	B

B	A
C	B

•Values within both of the diagonal pairs are equal to one another

There are only two grids that can be formed by the rotational and reflective symmetry of the original grid:

A	B
B	A

B	A
A	B

Table 3.7 shows the number of essentially different 2×2 grid arrangements formed, where the size of the intersection of the two candidate sets, between 1 and 8, is given. The table also shows how many solutions exist for each grid. Based on the cases above, each essentially different grid, shown in Table 3.6, may possess none, one or two *equal diagonals*; equal values within cells placed diagonally to one another in $k_{1,1}$ and $k_{2,2}$ or in cells $k_{1,2}$ and $k_{2,1}$.

Table 3.7: The number of grids counted with 0, 1 or 2 equal diagonally placed value pairs

Number of Solutions	1			2			3			4			5			6			7			8		
Number of Equal Diagonals	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
1 intersection	49	13	0																					
2 intersections	15	0	0	112	56	4																		
3 intersections	2	0	0	39	0	0	63	48	0															
4 intersections	0	0	0	0	0	0	6	0	0	26	40	8												
5 intersections	0	0	0	0	0	0	0	0	0	36	0	0	15	50	0									
6 intersections	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	24	12						
7 intersections	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	21	0			
8 intersections	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12
Total Per Number of Diagonals	66	13	0	151	56	4	69	48	0	62	40	8	15	50	0	15	24	12	0	21	0	0	0	12
Overall Total	79			211			117			110			65			51			21			12		

Table 3.8, based on the information of Table 3.7, summarises how many essentially different grids exist that possess none, one or two equal diagonals; equal values within cells placed diagonally to one another in cells $k_{1,1}$ and $k_{2,2}$ or in cell entries $k_{1,2}$ and $k_{2,1}$.

Table 3.8: Summary of the number of grids counted with none, one or two equal diagonals

Number of Equal Diagonals	Number of Unique Grids
0	378
1	252
2	36

In Section 3.2.3, the number of possible valid arrangements of values within a 2×2 grid was calculated using the diagonal pairs method. There are $x(x-1)(x^2-3x+3)$ valid ways of placing the values without duplication,where x is the highest value that may be used. This means that if the usual range of values, $1, \ldots, 9$ is used, there are 4,104 valid arrangement of values. It must be noted that this result does count a rotation or reflection of a given grid as an alternative, distinct grid.

Table 3.8 shows that in total, there are 378 essentially different grid arrangements where neither of the values placed in either of the diagonal pairs of cells are equal to one another, 252 essentially different grid arrangements where one the two diagonal pairs of cells contain equal values and only 36 essentially different grid arrangements where both diagonal pairs of cells contain equal values. So using the rules of Section 3.4.2 regarding counting rotations and reflections as different grids, there are $(378 * 8) + (252 * 4) + (36 * 2) = 4,104$. This total agrees with the number of valid arrangements that was calculated in Section 3.2.3. From Table 3.7, only $(66 * 8) + (13 * 4) = 580$ of these puzzle grids possess a unique solution, so are well-formed.

This method has proved successful for 2×2 puzzle grids, which is the smallest possible Kakuro grid. When grid size increases, there are more white cells and the patterns of white cells are no longer necessarily rectangular (since the corner cells may now be black for example). There may also be internal black cells within the puzzle grid. The placement of a value into a cell, the top-right cell for example, would no longer make the placement of adjacent values trivial, as was the case for 2×2 grids, meaning the enumeration of such grids would become far more difficult. This confirms that "The complexity of the problem

rises disproportionately with the size of the puzzle" [26].

In this chapter, bounds on the number of valid arrangements of values that can be placed within given grids have been considered. Such an analysis examined the underlying puzzle properties, especially the candidate sets associated with each run and the number of arrangements of values that could be used to validly complete runs. Chapter 4 describes standard methods for automating the solutions to problems and considers how such an approach may be implemented for Kakuro, hence enabling the evaluation of the usefulness of each approach. An approach that is able to incorporate both the fundamental puzzle constraints and, where possible, underlying puzzle properties is desired. Candidate sets are used to inform pruning conditions and a method of cell ordering during an automated approach to the solution of Kakuro puzzles in Section 5.5.

Chapter 4

Overview of Automated Problem Solving Approaches

Automated approaches to the solution of Kakuro puzzles can be placed into two categories. One category of approaches would use search algorithms, possibly along with heuristics and objective functions for optimisation. Such heuristics and objective functions would incorporate problem domain information so that the solver arrives at a valid solution. Alternatively, the secondary category would use similar methods to those used by a human solver, where the constraints of the puzzle (run-totals and non-duplication of values within runs) are considered in turn in some logical order. This ordering would mostly relate to the completion of cells for which only one possibility remains, until a valid solution is found.

In this chapter, standard methods for automating the solutions to problems are considered. Each general approach is described and consideration is then given to how the approach may be implemented for Kakuro. This enables the evaluation of the usefulness of each approach.

4.1 Exhaustive Search Techniques

Success has been achieved in the automated solution of many real-world problems through their representation as state space problems. Any problem that can potentially be solved using a formal search-based approach must have [53]:

- A state representation, including an initial state (or initial position of the problem), a goal state (or states) and intermediary states,
- A goal test, that indicates when a solution has been reached,
- A set of operators that map one state to another so as to produce successor states.

A problem is formulated in terms of a set of variables, or an ordering of objects, that relate to the domain of a specific problem (for example, in the Travelling Salesman Problem [41], a route must be constructed of cities to be visited in the shortest possible round trip; an ordering of cities represents one state). The state may be an *incomplete formulation* [41] of the variables or objects relating to the problem (such as a crossword puzzle in which only some of the clues have been filled in) or a *complete formulation* (such as a full list of cities for the Travelling Salesman Problem). Hence, the operators either set the value of a variable (or position an object), or they alter the previously assigned values of variables (or reorder the objects such as by swapping pairs of objects).

The states may be thought of as forming a *search space*, which is represented by a tree of states or nodes (Fig. 4.1). The root of the tree is the initial state, and the goal, or goals, reside in one or more positions further down the tree. Each link between states represents one legal application of an operator, *i.e.* a move. Each branch represents an ordered sequence of operators on the initial state, or one valid route through the space. An intermediary state, between the initial state and a goal state, can be thought of as a partial, or possibly even an 'incorrect', solution to the problem.

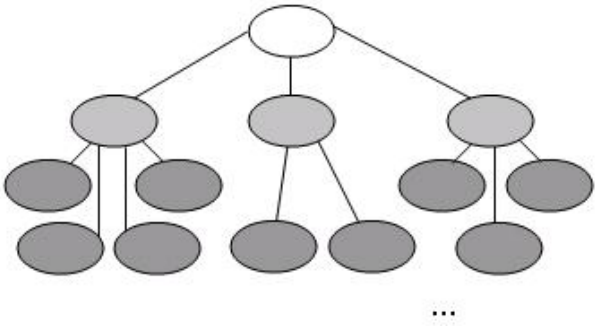

Figure 4.1: A generic search space

The aim is to arrive at a goal state (solution) through the application of the operators. This process, called *operationalisation*, creates a formal and manipulative description of the often informal original problem. The operations that map one state to another are part of a control strategy that should strive to cause *motion* [53], so that the search moves closer to a goal state. An infinite loop, for example, would cause a lack of motion so would be highly undesirable.

The most basic approaches to state-based search are *exhaustive*, or *uninformed*, search methods. Examples of this class of methods are *breadth-first* search and *depth-first* search [39, 53]. These are control strategies that dictate the order in which states are explored, *i.e.* added to the search space. In general, exhaustive search techniques, as expected, check all states within the search space until a goal state, or solution, has been found. The only way to ensure that a solution is the "best solution" is to examine the entire search space [41]. This approach alone may involve the exploration of an enormous search space, even for modest sized problems, and therefore requires a large amount of computation time.

Breadth-first search explores the search space one level at a time, enumerating every node on one level before descending to the next level. This control strategy is illustrated in Fig. 4.2(a) (which shows the first three levels of each of these two search approaches in a generic search space) in which the consecutive letters indicate the order of enumeration. Depth-first search explores one branch of the search tree until there are no further successors and then returns to the previously explored state (as illustrated in Fig. 4.2(b)). If the search space is extremely deep, or infinite in depth (having operators that can be applied to any state), a depth cut-off can be employed [39]. This cut-off imposes a maximum limit on the depth of the search before returning to a node one level higher up (a strategy sometimes also referred to as depth-limited search [55]). Moving one level higher up, *i.e.* returning to a parent node, is termed *backtracking*.

Implementations of the depth-first search approach are generally more efficient than breadth-first search approaches because only the states in the currently explored branch need to be stored, although for some puzzles, the search may become trapped exploring a fruitless path.

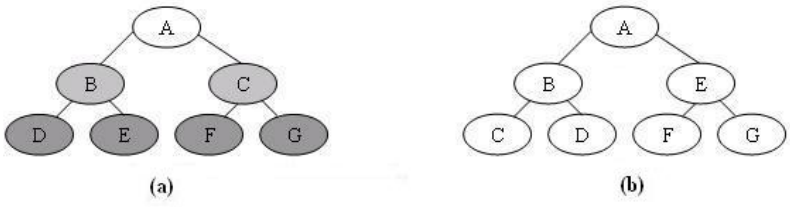

Figure 4.2: The first three levels of breadth-first (a) and depth-first (b) search

It may not always be necessary to descend each branch until either the end of the branch is reached or a solution is found. If the depth-first algorithm can employ problem domain knowledge to determine that a goal cannot lie further down a branch, *i.e.* a "dead end" has been reached, it would be pointless continuing the exploration of this path and so the algorithm backtracks to the previously-explored state. This is also referred to as *pruning* the search space.

4.1.1 An Exhaustive Search Approach for Kakuro

A Kakuro puzzle has a form that is suitable for a state representation, either as a complete state formulation or an incomplete state formulation. In the former, the puzzle would be filled with an initial set of values and in the latter, the cells would be filled one by one with each iteration of the search. The choice of formulation dictates the construction of the operators. Either formulation leads to a clearly defined search space, which corresponds to the set of all possible partially-filled and/or filled grid arrangements. (This allows a variety of search techniques to be used in order to arrive at a valid solution state from the initial state; this section will focus on only the application of exhaustive methods.)

In puzzles such as Sudoku, the solver knows *a priori* how often each value occurs, encouraging the construction of an initial state that includes the placement of all values, albeit in the wrong order. Search might then proceed by swapping pairs of values. In Kakuro however, it is not initially known how often each value will occur. It may therefore be convenient to implement an exhaustive approach using an initial state which is empty. Each iteration of the search algorithm assigns a value to one cell. A state therefore is simply a partial assignment of values to the cells within a puzzle grid. An operator transforms one state to

another by adding a value to a particular cell. This is illustrated in Fig. 4.3 below, which indicates the nine possible states for the first move.

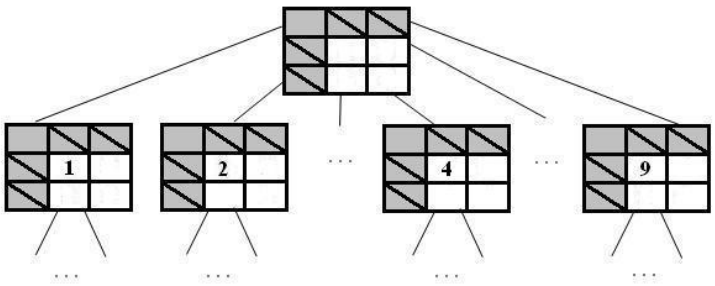

Figure 4.3: An example implementation of the first search level of an exhaustive search

For this implementation, the depth of the tree is equal to the number of cells within the puzzle. The solution must therefore lie on the deepest level. The control strategy (breadth-first or depth-first) successively applies operators until the deepest level has been reached. The goal test determines whether the puzzle has been solved, or whether it is necessary to backtrack. Therefore, an exhaustive control strategy may potentially fully enumerate the search space in order to locate the solution. Since Kakuro puzzles should be well-formed (and so possess one solution), the algorithm will generally terminate prior to full enumeration. However, full enumeration would be necessary to ensure uniqueness of solution.

For grids having large numbers of cells, exhaustive methods will inevitably be inefficient and time consuming due to the number of states that would need to be considered. Although seemingly effective while a smaller grid is under investigation, the difficulty of enumerating all states and the number of states within the search space can increase substantially (Section 3.4.2). Pruning of fruitless branches would be of benefit. In Kakuro, the placement of many values will lead to violations of the puzzle constraints – a duplicate value in a run, or an exceeded or under-target run-total on completion of the final cell of a run. Such violations could be used to prune an evidently fruitless branch.

4.1.2 Evaluation

Exhaustive search methods are certainly appropriate for problems having a relatively small search space.

> "Although it is possible in principle to solve any problem in this way, in practice it is not..." [52]

Since a large number of states may have to be checked, the basic exhaustive approach would be adequate for Kakuro puzzles with smaller grids, or those that use a smaller range of values, but it would be very time consuming and inefficient for most other puzzles, without considerable pruning of the search space. Problem domain knowledge relating to puzzle violations (namely run-total and duplication constraints) can be used effectively to prune fruitless branches of the search space, resulting in far fewer states being explored.

Since all solutions lie at the deepest level within the search space, an implementation of a depth-first search approach would be more efficient as fewer states would have to be stored (at most, the number of states equal to the depth of the tree would be stored). Therefore, a depth-first approach with the pruning conditions of backtracking would seem to be promising for the solution to Kakuro puzzles, providing suitable pruning is implemented.

4.2 Local Search Techniques

Using the definitions of *intermediary states*, *initial state*, *goal states* and *operators* of Section 4.1, attention is focused here on the *local neighbourhood* of some current state. The local neighbourhood of a state is found by applying a valid operator to the current state to derive successor state(s). Hence, the current state of the search is expanded and the merit (or score) of each successor is evaluated. The best such successor state is then selected for further subsequent expansion [39].
During a local search procedure [41]:

1. An initial solution is constructed, often at random. This is termed the *current* solution, and represents the initial state. Its merit is evaluated using some *objective function*,

2. A transformation is applied to the current solution so that *new* successor solution state(s) are generated. An implementation may store these successor states within a

'queue'. Their merits are evaluated using the objective function,

3. If one of the new solution states is better than the current solution, then that new solution becomes the current solution. Otherwise, the new solutions are discarded,

4. Steps 2 and 3 are repeated until no improvement is possible as a result of the transformation.

A local search optimization approach [53] uses an element of "scoring"; it relies on a concept of alternative states being "better" or "worse" than one another. A partial solution may therefore be regarded as a "bad" solution that can be improved. Such scoring is performed by an objective function, which is devised using suitable problem domain knowledge. The objective function is a mechanism to guide the solver through the search space by determining an efficient path that begins at an initial state. The function may be designed to produce a higher score (a maximising function) to indicate an improved solution or a lower score (a minimising function).

In addition to problems associated with a potentially large search space (arising when there is a large number of white cells present in the puzzle grid), local search suffers from two other difficulties – it can become trapped in local optima and search spaces can include plateaus. There will inevitably be some points within the search space where a score appears to be the best possible, since no local move appears to improve the current state score, particularly for problems where more than one solution exists. The scores assigned to each successor state are worse than that assigned to the current (parent) state, meaning that the algorithm has become trapped at a local optimum. Note that such a local optimum may be a maximum (where a high score is desired but scores of successor states are all lower than that of the current state) or a minimum (where a low score is desired but scores of successor states are all higher than that of the current state). A better, *global* solution would hence lie elsewhere in the search space [41]. Fig 4.4 illustrates such cases. In addition, many states may be assigned identical scores, particularly by a weak objective function, so there is uncertainty about which successor state should be explored next. This may be thought of as creating plateaus in the search space. Paths through the search space may not lead to a goal in a desired, optimal time and may not lead to a solution at all.

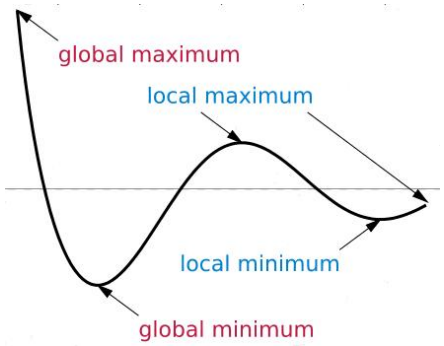

Figure 4.4: An example of local maxima and minima

Due to the above difficulties, the optimal value reached often depends on the initial state chosen; repeating the approach with an alternative initial state may result in an alternative (local) optimum, which also may or may not be the global optimum. Assuming that a given local optimum is not itself the global optimum, there is no information about the amount by which this local optimum varies from the global optimum. Michalewicz and Fogel [41] believe that in general, it is *not* possible to provide an upper bound for the computation time required to find the global optimum, but suggest that to attempt to further avoid becoming trapped in local optima, the neighbourhood may be enlarged. Alternatively, heuristics could be used, which generally improve efficiency of a search process but at the expense of the solution quality. Such an approach will almost always find a *very good* solution but is not guaranteed to find the best solution [41].

4.2.1 A Local Search Approach for Kakuro

Whereas for many problems it can be difficult to formalise a scoring mechanism, the numeric nature of Kakuro puzzles initially suggests that the puzzle could allow the use of a fairly simple scoring system for the objective function within a local search approach. Such puzzles may therefore be appropriate for a local search approach.

In Section 4.1.1, Kakuro puzzles were found to have a form that is suitable for a state representation. Such representation may employ either a complete state formulation or an incomplete state formulation. In the latter, the cells would be filled, one by one, with each iteration of the search, meaning that the grid is only partially filled at some given iteration. Empty cells will complicate the scoring of the state by the objective function; it would be

unclear how a scoring mechanism should distinguish adequately between two states both having mostly empty cells, for example. An incomplete state formulation is therefore not ideally suited to a local search approach. By comparison, it seems more intuitive to distinguish between two completely filled grids, which may be thought of as representing "poor" or incorrect solutions. An objective function may more easily be constructed for a complete state formulation.

In a complete formulation, the puzzle would be filled with an initial set of values. This set of values would not generally have the correct number of occurrences of the values 1 to 9 within the grid and it would be very unlikely that any run-totals are satisfied. Operators might then replace values within cells of the puzzle. Fig. 4.5 shows the first level of a search space for a small Kakuro using this problem description, showing the successor states of an initial state containing randomly assigned values:

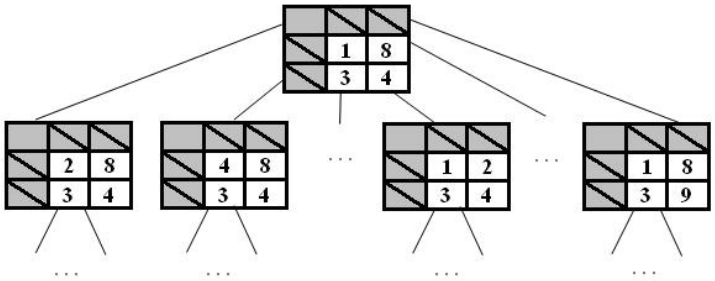

Figure 4.5: An example of a first level of a search space for Kakuro in a local search approach

Note that if the objective function was omitted, then this would be identical to an exhaustive search, explained in the previous Section. Such exhaustive search control strategies would cause motion, but may involve a high time overhead, especially due to the fact that the same state may be investigated more than once.

The proposed complete state formulation provides a mechanism for traversing a search space that includes the goal state (thought of as the global maximum) possibly at multiple positions within a potentially very large, finite search space. The success of this approach depends on determining an objective function that can reliably move toward a goal state.

Finding a useful objective function to be used during the solution of Kakuro puzzles using this approach is highly problematic. The amount of problem domain information relating specifically to Kakuro is limited – only how closely the current run sums match specified run-totals. Therefore, the limited amount of information that can usefully be incorporated into an effective objective function may be detrimental to the effectiveness of such a function. Inevitably, many states at distances from the goal state will map to the same objective function score. Similarly, large numbers of states in a given neighbourhood may have identical scores. There would therefore be an increased likelihood of the method becoming stuck in plateaus in the search space [39, 53]. Similar difficulties have been reported in a local search optimisation approach to the solution to Sudoku puzzles [33]. Also, since each value in a particular cell can be replaced by up to eight alternative values, the search space growth would be combinatorially explosive.

4.2.2 Evaluation

The numeric nature of Kakuro puzzles make this approach seem feasible. However, the limited amount of puzzle domain information that is available, relating to how closely the current run-totals compare with the goals, does not suggest an effective objective function that will reliably move to a better state within the search space. Solutions of larger puzzles typically require more iterations than their smaller counterparts before a solution is located in the search space. In some implementations, a queue would then have to store a very large number of partially filled states. (However, alternative implementations may only dynamically recalculate the immediate neighbouring states each time the current state changes, discarding any previous storage.) There are not strong reasons to favour local search approaches when a solution may be found using an exhaustive approach with pruning.

4.3 Metaheuristic Search Techniques

Metaheuristics are used to solve various computational problems by adding a high-level algorithmic approach that guides existing control strategies and heuristics in a search for feasible solutions. They use concepts derived from artificial intelligence, biology, mathematics, natural and physical sciences to guide and modify heuristics to produce solutions beyond those that are normally found in a quest for optimality. Metaheuristics are generally applied

to problems for which there is no satisfactory problem-specific algorithm, or when it is not practical to implement such a method [19, 55]. Specifically, metaheuristic approaches might be employed to overcome the limitations of the objective function.

Examples of metaheuristics include tabu search [41, 52], genetic algorithms [43, 55] and multiple hill-climbing, sometimes referred to as simulated annealing [41], where a local search approach is simultaneously run from several initial points to attempt to overcome local optima. The best *overall* outcome is returned following the completion of the simulated annealing approach. A tabu search approach employs prohibition-based techniques that complement basic search algorithms [52]. Again using the definitions of Section 4.1, an operator makes an optimal change to the current state; the change that produces the most improvement to the current score of an objective function is selected. One implementation of the *tabu* aspect may then memorise the most recent operations and prevent their re-use for a period, *i.e.* these become *tabu points* that are to be avoided while making decisions about selecting the next solutions during the next x iterations. This element of "memory" forces the search to explore new areas of the search space, possibly avoiding becoming trapped at a local optima. Tabu search works on complete solution states. Compared to more classic approaches, there are more parameters to become concerned with, each of which would need to be "fine tuned" to the problem, posing yet more difficulties [41].

Genetic algorithms consist of populations of "chromosomes", which represent states of some problem search space. Chromosomes within a given population have evolved from those within an initial population which may be randomly generated. The simplest form of genetic algorithms consist of three categories of operators [43]:

- Selection – this operator selects the "fittest" chromosomes for reproduction,
- Crossover – these operators exchange parts of two chromosomes at a certain point to create two offspring, and are usually assigned a probability of occurrence,
- Mutation – these operators randomly change a chromosome, and are usually assigned a probability of occurrence.

The purpose of crossover is to pass on to offspring advantageous properties of both parent chromosomes, preserving beneficial aspects of those solutions. Mutation, and to some extent crossover, both assist in escaping local optima, described in Section 4.2. For any specific

problem, there are a number of details to specify [43]:

- How a chromosome will be represented and how it will correspond to the actual problem representation,
- The fitness function, which evaluates the fitness of each chromosome in a generation in order to decide which are to be used to evolve into the next generation,
- The probabilities of occurrence of both mutation and crossover,
- The exact details of how mutation and crossover will occur,
- The size of the population,
- The number of generations to be used before accepting the best (fittest) solution so far as the overall solution.

At the end of a *run* (the entire set of generations), there are often highly fit chromosomes present in the generation which correspond to the best solutions to the given problem.

4.3.1 A Metaheuristic Approach for Kakuro

Consider first a tabu search implementation of Kakuro. An initial state might be constructed by a random assignment of values to cells. The application of an operator makes a change to one cell within the current state. The selected change would be the optimal change, that which makes the most improvement to the current score, given by an objective function. The tabu aspect may, following a change to a cell, restrict any further changes to that cell for the next x iterations, *i.e.* the cell is tabu. This element of "memory" forces the search to explore new areas of the search space, possibly avoiding becoming trapped at a local optima. This approach may alleviate the issues of having a poor objective function, however, the objective function would still be used to find a "better" solution within the given neighbourhood. Like during a local search approach, many states would still have the same score, increasing the likelihood that the scores of the solutions in the pool would converge around some local optimum, or plateau, requiring mutation to escape.

Due to the numeric nature of Kakuro puzzles, a genetic algorithm approach may be effective in automating their solution. Each puzzle state (chromosome) may be represented

by a bit string where some mapping function exists to map each bit string to the familiar grid structure. The length of the bit string relates to the number of white cells within the grid, and is therefore constant. Each chromosome encodes the values assigned to all cells within a given state. Crossover operations would be required to "breed" two chromosomes at some specified point, relating to a cell within the grid. (Hence, crossover would, from each parent, take the current solution to half the cells.) Mutation operations would change a bit within a chromosome. Probabilities of occurrence would be assigned to mutation and crossover operations. A fitness function would assign some score to each chromosome within a generation to decide which will be used to "breed". However, like an objective function in a local search techniques, a fitness function may not be effective since there is little puzzle domain information that can be used to develop such a function. Many chromosomes may therefore receive the same score.

4.3.2 Evaluation

Metaheuristic approaches typically add more parameters. Such parameters will inevitably need to be "fine tuned" to the problem in hand which introduces a new question: "how are parameters going to be assigned in order to derive an optimum solution to the problem in hand?" [41].

It is worth noting that such techniques may still be susceptible to any failings of the underlying control strategy or heuristic. For example, a tabu search approach, like local search, can still be liable to become trapped in plateaus despite the "memory" aspect. The effectiveness of a genetic algorithm is likely to depend on whether a crossover point can be chosen such that good solution characteristics of the parent chromosomes are preserved. It may be possible for some puzzles to choose the crossover point (a single cell) such that a sufficient number of unbroken runs are located either side of the crossover. However, consider a run, r_l of length $\mid r_l \mid$: as each cell is considered to belong to two runs, this run will intersect with $\mid r_l \mid$ other runs. The author therefore believes that, in general and particularly for large puzzles, it would be difficult to avoid breaking many runs.

Metaheuristic approaches generally involve high processing overheads, and can be inefficient. The author takes the view that an exhaustive implementation, with pruning, can reliably be

used to locate the solution to a given puzzle in the search space, and this simpler approach has been pursued in preference to the more elaborate metaheuristic approaches.

> "The more sophisticated the method, the more you have to use your judgement as to how it should be utilised." [41]

4.4 Constraint Satisfaction Techniques

Constraint based approaches view a given problem as a collection of variables whose values must be assigned such that various stated constraints are satisfied. Constraints sometimes reduce the size of the search space, hence allowing a solution to be found more quickly. Such problems are not concerned with optimisation issues ("best" or "worst" solution *etc.*) but instead seek any feasible solution [41]. A constraint satisfaction problem uses general purpose rather than problem specific heuristics to solve problems. In [49], a solution to a Sudoku puzzle is found using linear programming by finding the unique non-negative solution to the system of linear equations. Binary integer programming, a special case of linear integer programming, is one form of a constraint satisfaction problem.

4.4.1 Constraint Satisfaction Techniques for Kakuro

Kakuro puzzles possess explicit puzzle constraints that can be used to find a solution, the validity of which can easily be checked. These puzzle constraints, namely the summation requirement of values to a specified run-total and non-duplication of values within runs, make Kakuro puzzles seemingly appropriate for a constraint-based approach to a solution.

Such a formulation of the puzzle has previously been presented by the author [14]. In that formulation, ten binary decision variables, $A_{i,j,h}$, are associated with every cell in the $n \times m$ grid, where i and j specify the row and column positions of the cell, and h specifies an available value for assignment to the cell. (Black cells cannot accept a numerical value so are forced to accept the value zero.) Of these ten binary variables per white cell, only the one for which h matches the value currently assigned to the cell will be set equal to one. The other nine variables associated to that particular cell would then be set equal to zero.

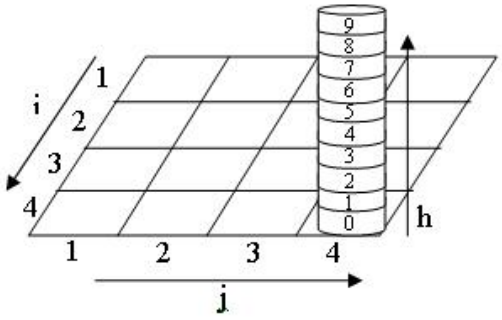

Figure 4.6: A visualisation of a B.I.P. approach for Kakuro

A useful visualisation of this approach is shown in Fig. 4.6; each cell may be thought of as corresponding to a "tower" having ten "floors". Each floor represents a value that may be assigned to the cell. If, for example, a nine is placed into the cell at the bottom-right corner of the 4×4 puzzle grid of Fig. 4.6, the ninth floor would become "lit". This corresponds to a decision variable associated to that cell, $A_{4,4,9}$, being assigned the value one. The black "clue" cells cannot be assigned a digit so in these cases, a constraint would force floor zero to be lit.

Formally, puzzle constraints and trivial constraints governing the binary assignment of values to variables $A_{i,j,h}$ are expressed explicitly:

- Only one value can be assigned to any one cell;

$$\sum_{h=0}^{9} A_{i,j,h} = 1 \qquad 1 \leq i \leq n \qquad 1 \leq j \leq m$$

- No numerical value in the range $1, \ldots, 9$ can be added to a black "clue" cell, and so black cells must be "assigned" the value zero. Hence for any black cell $k_{i,j}$:

$$A_{i,j,0} = 1$$
$$A_{i,j,h} = 0 \qquad h \neq 0$$

- Conversely, white cells can only accept values in the range $1, \ldots, 9$. Hence for any white cell $k_{i,j}$:

$$A_{i,j,0} = 0$$

- Within each run, there must be no duplication of digits. Hence for a run in *row* i of grid K that begins in column j_s and ends in column j_e:

$$\sum_{j=j_s}^{j_e} A_{i,j,h} \leq 1 \qquad 1 \leq h \leq 9 \qquad 1 \leq i \leq n$$

and for a run in *column* j of grid K that begins in row i_s and ends in row i_e:

$$\sum_{i=i_s}^{i_e} A_{i,j,h} \leq 1 \qquad 1 \leq h \leq 9 \qquad 1 \leq j \leq m$$

- For each run, r_l, the digits must sum to the required run-total, termed t_l. Ten decision variables are associated to each cell within the puzzle grid, but only one of these, the one with h value matching the digit in the cell, is equal to 1. Hence for a horizontal run in *row* i of grid K that begins in column j_s and ends in column j_e:

$$\sum_{j=j_s}^{j_e} \sum_{h=1}^{9} hA_{i,j,h} = t_l \qquad 1 \leq i \leq n$$

and for a vertical run in *column* j of grid K that begins in row i_s and ends in row i_e:

$$\sum_{i=i_s}^{i_e} \sum_{h=1}^{9} hA_{i,j,h} = t_l \qquad 1 \leq j \leq m$$

The solution is indicated by the collection of binary decision variables, $A_{i,j,h}$, that are set to 1, showing which value h should be assigned to the cell at row i and column j. Such assignments must satisfy all puzzle constraints which were explicitly declared to the program used: XPress MP (a suite of optimisation packages).

4.4.2 Evaluation

Tests were performed on a Viglen Intel Core 2 Duo processor 2.66GHz, with 2GB RAM. Programs were developed in XPress MP (2008 release). Xpress-MP is a suite of mathematical modeling and optimization tools used to solve linear, integer, quadratic, non-linear, and stochastic programming problems. Three puzzles from each grid size between 2×2 and 14×14 and additionally a 29×29, 34×34 and 39×39 sized grid were investigated. The results for this approach showed that this approach arrived at a solution quickly for all small puzzles tested ([14]) and also for all those from an additional test set that had grid size ranging up to 29×29. In fact, a solution is found in one second or under for all puzzle grids tested (ranging in size up to 29×29). The software failed to find a solution for the

largest available grids, 34×34 and 39×39 in size, where there are large number of variables present, due to a lack of computer memory. This is a computer limitation rather than a software limitation so it is thought that with the use of a more powerful machine, a solution could be found, given enough resources.

Although constraint based approaches can be extremely effective, they can be unable to locate all solutions to a given puzzle grid (and hence cannot determine whether a puzzle has a unique solution). They may also fail to find a solution for large grids where there are a high number of variables present without using some kind of search or trial and error [53]. Since well-formed Kakuro puzzles possess only one solution, this would not ordinarily be a problem. However, this method could not therefore be used to verify that a given puzzle does indeed have a unique solution. A 5×5 puzzle that possesses two solutions is shown in Fig. 4.7; depending on the choice of dummy objective function, this approach would only output one of these two solutions since there is no notion of a "better" or "worse" solution.

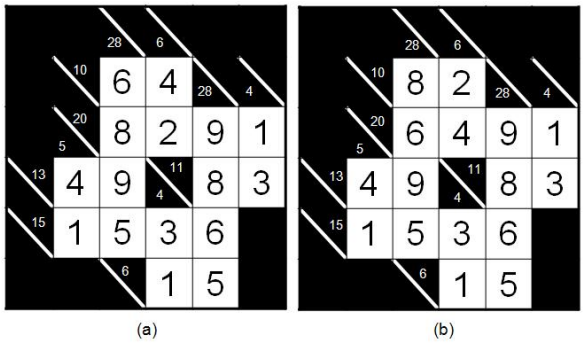

Figure 4.7: A 5×5 Kakuro Puzzle that possesses multiple solutions

The preference for a method of automated solution that ensures solution uniqueness, and which is extendible to solutions of puzzles of large sizes, leads the author to reject constraint satisfaction approaches. However, it is thought that they may be worth further investigation.

An alternative implementation of the above binary integer programming approach toward the automated solution of Kakuro puzzles may have assigned variables to only the white cells. This may have increased efficiency due to the lesser number of variables and constraints in use. Another constraint satisfaction approach has subsequently been applied to

Kakuro puzzles [59]. In that approach (explained in more detail in Section 2.2.5), integer programming was used; each cell is assigned a finite domain variable with values in the range 1 to 9. In a similar way to the binary integer programming approach of Section 4.4.1, puzzle constraints are formally declared.

4.5 Summary of Approaches

A local search approach is rejected in favour of an exhaustive approach with pruning. A declared aim for this work is to identify and analyse properties of Kakuro puzzles, and so a method for automating the solution of Kakuro puzzles that employs problem domain information directly is preferred. However, the amount of problem domain information relating specifically to Kakuro seems to limit the construction of an effective objective function. There is a likelihood of the method becoming stuck in local optima and plateaus [53], where many states have the same score.

Metaheuristics, such as tabu search and genetic algorithms, are often elaborate schemes and can involve high processing overheads for the generic metaheuristic implementation. In terms of genetic algorithms, a similar problem to that of the objective function of a local search approach may be encountered; a fitness function may not be effective since there is little puzzle domain information that can be used to develop such function. (This issue may also arise during implementation of a tabu search.) Many chromosomes may therefore receive the same score. The choice of a suitable crossover point, to preserve good solution characteristics of the parent chromosomes, may also be problematic. Metaheuristic approaches are therefore not pursued for Kakuro puzzles, as effective, simpler approaches seem to be available.

Although constraint based approaches show great promise, they are unable to locate all solutions to a given puzzle grid (and hence cannot determine whether a puzzle is well-formed) and fail to find a solution for large grids where there are high numbers of variables present.

An exhaustive approach places all possible values into all possible cells and so is guaranteed, eventually, to find all solutions. Such an approach may be adequate for Kakuro puzzles that have a small grid, or for puzzles that require the use of a small range of values. The addition

of backtracking to such approaches, where the remainder of the current branch of the search space is pruned, eliminates the need to explore further parts of the search space where a solution would definitely not lie. It is expected that the use of pruning within a depth-first backtracking implementation will lead to an efficient approach to the automated solution of Kakuro puzzles and will enable puzzle properties to be highlighted. An implementation of a depth-first backtracking solver is produced and tested in Chapter 5.

Chapter 5

Automation of Solution

An approach that takes direct advantage of the problem complexity characteristics of Kakuro puzzles, notably the permutations of the values that may legitimately be assigned to runs, is desired. Chapter 4 identified the advantages of using a depth-first approach to the automated solution of Kakuro puzzles. In particular, this approach is efficient in storing partial solutions because only the states in the currently explored branch need to be stored. Further, this method is suited to problems in which problem domain information can be used to determine that a goal cannot lie further down a branch within the search space, as is the case with Kakuro. The pruning of such a fruitless branch is achieved through the use of backtracking.

A backtracking algorithm, employing a depth-first approach to examining the search space, is a form of exhaustive search. Suitable heuristics, *i.e.* "rules of thumb" or functions defined to direct an algorithm through a search space via the cheapest path to a goal state, are used to guide the backtracker. Such heuristics will exploit the features of the problem domain in order to reduce time spent examining a search space [53]. Effective pruning conditions can be determined to reduce the number of states that have to be investigated within the search space. Reducing the number of states will reduce the overall time taken to find a solution. However, the introduction of heuristics and pruning routines come at a cost in processing time. In this chapter, two backtracking algorithms are described – an implementation through the use of a stack (Section 5.1) and a recursive implementation (Section 5.4). Pruning rules and heuristics are incorporated into both algorithms to improve the solution time.

5.1 An Implementation of a Stack-Based Backtracking Solver

In this section, an implementation of a basic stack-based backtracking algorithm for the automated solution of Kakuro puzzles is explained. Modifications to this algorithm are also explained in Section 5.2. Its performance will be evaluated in Section 5.3.

The approach of Algorithm 5.1 begins with an empty grid, implemented using an array within the Java programming language, and iteratively attempts to assign values to each white cell in turn, beginning with the lowest numerical value, and beginning the placements from the array location corresponding to the top leftmost white cell. It follows a depth-first [53] enumeration of the search space, favouring the assignment of low numerical values. Tests within the algorithm ensure that some fruitless paths, specifically when a constraint violation occurs, are avoided. Hence some pruning of the search space is employed. An apparently successful assignment of a value to a cell (one which does not violate puzzle constraints) will result in the current grid ("Current_State") being pushed onto a user-defined stack, implemented using an "arraylist" within the Java programming language. Since the stack holds puzzle states, this is actually implemented as an "arraylist" of arrays within Java. Violations of the puzzle constraints - a duplicate value in a run, an exceeded run-total or an under-target run-total where all possible values have been considered for the final cell of a run - will result in the algorithm backtracking, and popping the last successful grid state from the stack. This implementation of the stack only stores incomplete states, that are apparently valid, along one branch of the search space, thus avoiding the memory based issues which can arise in search approaches in which all valid partial states encountered are stored (for example in the queue of a breadth-first search approach, where all apparently valid partial states would need to be stored from each and every level of the search space until a solution is found at a level that may be considerably low [53]). An iteration count is incremented each time an attempt is made to assign a value to a cell, and is used as a measure of algorithm performance in Section 5.3. This implementation has not been pursued. An alternative implementation, instead of pushing partially filled states to the stack, may push only the part containing cells that have been assigned values. Empty cells would not

then be stored multiple times within the stack.

While this approach is ideal for smaller puzzles (as will be demonstrated in Section 5.3), the algorithm may be required to perform a great deal of backtracking in larger puzzles. The algorithm may reach the final cell before a violation is detected. This difficulty makes the addition of further components desirable. The following section proposes heuristics and additional pruning conditions that can be added to this algorithm.

Algorithm 5.1: Stack-Based Backtracking Algorithm

Initialise stack and global Iteration_Count, Puzzle_Run_Cells, Puzzle_Runtotals and Solution_Stack.
Current_State becomes the InitialState.
Add Current_State to stack.
Current_Cell is set to be the first available white cell.
Current_Value = 1.
while empty white cells exist **do**
  Place Current_Value into Current_Cell.
  Increment Iteration_Count.
  Determine runs in which Current_Cell resides, and corresponding run-totals.
  if [no duplicates in runs] and ([run-total(s) not exceeded] or [run(s) completed correctly]) **then**
    Push Current_State to stack.
    if empty white cells exist **then**
      Current_Cell becomes next available cell.
    end if
    Reset Current_Value to 1.
  else if ([runs under target run-totals] or [duplicate in run(s)]) and [Current_Value <9] **then**
    Current_Value = Current_Value +1.
  else
    Pop state from stack to become Current_State.
    Current_Cell becomes previous cell.
    Current_Value becomes value within Current_Cell.
    while Current_Value = 9 **do**
      Pop state from stack to become Current_State.
      Current_Cell becomes previous cell.
      Current_Value becomes value within Current_Cell.
    end while
    Current_Value = Current_Value + 1.
  end if
end while
Output Current_State as solution.

5.2 Modifications to the Backtracking Solver

In this section, modifications to the Stack-Based Backtracking Algorithm of Section 5.1 are proposed. The results of using the backtracking algorithm with and without such modifications are presented and analysed in Section 5.3.

5.2.1 Run-Based Cell Ordering

It is proposed here that the path taken through the search space be guided by consideration of how many valid arrangements of values can be used to satisfy each run. This *cell ordering* heuristic favours the completion of cells in runs having fewest valid arrangements. By implementing this heuristic, a reduction may be achieved in the maximum amount of backtracking required due to incorrect assignments to cells that are considered near the start of the search process. Those cells in runs having most potential valid arrangements will be considered later, tending to push the consideration of cells requiring most backtracking to a deeper level in the search space, when most other cells will have already been assigned a value.

As an example, a run-total of 6 over two cells can be filled using the tuples (1, 5), (5, 1), (2, 4) and (4, 2). (The tuple (3, 3) would be invalid due to the non-duplication constraint.) Hence this run can be filled in four different ways and so would be assigned a score of four. A *look-up table* is constructed using the generating function of Section 3.3, which explicitly states how many distinct ordered arrangements of values exist for each run-total t_l over each possible run-length $\mid r_l \mid (\forall r_l \in r)$. Scores required are taken from this look-up table.

As this approach uses calculations based on entire runs, rather than on single cells, a cell inherits the lowest number of choices for the two runs in which it resides. This represents an upper bound for the actual number of choices for that cell. It can be noted that a more accurate measure is to be found in the intersection of the arrangements in runs, as outlined in Section 3.2.2 and Section 3.4. The production of a heuristic that is based on the values that can actually be placed into a cell, namely the values present in the intersections of the two candidate sets belonging to the two runs, is documented in Section 5.5.2.

5.2.2 Value Ordering

This heuristic favours the assignment of values in the range $1, \ldots, 9$ in reverse order, essentially being based on the "assumption" that puzzles will be solved more quickly in this manner. Clearly, all values are equally likely to be the content of a cell of a puzzle solution, in a general sense; the actual likelihood of, for example, a 1 or 9 appearing more frequently in a solution will be puzzle-specific. This is a poor heuristic, but no worse in general than the reverse assumption. Hence the heuristic provides a useful test of the performance of the algorithm, when measuring the results of many puzzles. A puzzle having several high values in cells considered at the start of solution will probably solve more quickly when using this heuristic.

5.2.3 Decisive Value Ordering

While the usefulness of reversing the ordering of values (Section 5.2.2) is highly puzzle-specific, it may be possible to produce a value ordering, based on puzzle domain knowledge, that reduces puzzle solution time in general.

Some cells within a given puzzle may benefit from being assigned values from the range $1, \ldots, 9$ in reverse order whereas some cells, particularly those in runs having a low run-total, would require more iterative exploration if the range of values were used in this alternative order. For each "Current_Cell", under consideration, an average cell value is dynamically calculated, providing an indication of whether the assignment ordering of values should be reversed. This calculation is based on both runs, r_{l_1} and r_{l_2} ($1 \leq l_1 < l_2 \leq p$ where p is the number of runs contained in the puzzle) with corresponding run-totals t_{l_1} and t_{l_2}, in which the cell resides, such that:

$$a_{i,j} = \frac{t_{l_1} + t_{l_2}}{\mid r_{l_1} \mid + \mid r_{l_2} \mid} \tag{5.1}$$

The algorithm uses the range of values $1, \ldots, 9$ if $a_{i,j}$ is less than 5 but uses the values in reverse order if $a_{i,j}$ is greater than or equal to 5.

5.2.4 Projected Run Pruning [P.R.P.]

Stack-Based Backtracking Algorithm 5.1 of Section 5.1 checks for invalid assignments to a run on the completion of that run. This will still allow poor choices of values to be placed

at the beginning of a run, such that the run-total can not be met with legitimate value assignments in the remaining cells. As an example, consider a run of 5 cells having the run-total 35. A placement of 1 in the initial cell will seem legitimate, but even the assignment of the largest values to the remaining cells - 9, 8, 7 and 6 - will only lead to a total of 31. In such a case, considerable processing time would be wasted attempting to fill the remaining cells, until the Backtracking Algorithm eventually places a value larger than 4 in the initial cell. Each time an assignment is made, by considering whether a run can possibly be completed to meet its total, fruitless branches of the search space can be pruned.

Additional validity checks are added to the Backtracking Algorithm of Section 5.1. On assigning a value to a cell in a run that still possesses unassigned cells, a calculation is performed of the sum of the largest possible values that may still legitimately be added to the remaining cells of that run, ensuring that this calculation does not include any duplicated high values already present in the run. If this sum yields a run-total at least matching the specified run-total for that cell, the backtracker continues, otherwise this branch of the search space is pruned and backtracking occurs.

Additionally, if only one cell remains unfilled within a run, a check is performed to calculate the difference between the run-totals of both corresponding runs and their current totals. If this difference cannot be met without assigning a value to the remaining cell that is already present in either the horizontal or vertical run in which the cell resides (hence causing a duplication violation), then this branch of the search space is pruned and backtracking occurs. The remaining value must be no larger than the lowest of the two run-totals that correspond to the cell. Otherwise, the algorithm continues, and the *required* value is assigned to the cell, in order to complete the run validly. Importantly, the algorithm only performs this check if the required value is greater than or equal to "Current_Value" to avoid possible unbreakable cycles. This pruning should further reduce the number of puzzle states that will need to be considered and hence should, in general, decrease the time taken to obtain a solution to a given puzzle.

5.3 Results of Modifying the Backtracking Algorithm

There is no published work on exhaustive or local search approaches to solving Kakuro with which to compare the results presented in this chapter, in order to evaluate the effectiveness of the backtracking solver, described in Section 5.1, and the modifications described in Section 5.2. The results obtained using the modified approaches will be compared to results obtained using the Stack-Based Backtracking Algorithm alone, for specific puzzles of varying sizes. Tests were performed on a Viglen Intel Core 2 Duo processor 2.66GHz, with 2GB RAM. Programs were developed using Java platform 1.5.0_06 within Oracle Jdeveloper 10.1.3.3.0, executed in the J2SE runtime environment. For timings, each puzzle is run ten times in succession and the best time recorded.

Initial experimentation focused on establishing the relative and general effectiveness of the methods proposed above, and results are shown in Tables 5.1 to 5.6 below. These tables present results for Stack-Based Backtracking Alone (Algorithm 5.1 with no modifications), Run-Based Cell Ordering (Section 5.2.1), Value Ordering (Section 5.2.2), Decisive Value Ordering (Section 5.2.3) and Projected Run Pruning (P.R.P.) (Section 5.2.4). Few puzzles of small size were available for testing, but those tested were deemed sufficient to examine the methods and to demonstrate the puzzle-specific nature of their effectiveness. A test set consisting of ten puzzles from each size grouping between 2×2 and 10×10 inclusive is therefore used. Tables 5.1 to 5.4 show the minimum, maximum, median and average number of iterations required (explained in Section 5.1) for each size grouping. Tables 5.5 and 5.6 show the median and average solution times for each size grouping, measured in milliseconds. The median statistic is used as an alternative to the mean, as it is less likely to be skewed by very large or very small anomalous results.

Some puzzles within the same size grouping took a very large or very small number of iterations compared to the average and median statisic. This due to the highly puzzle specific nature of Kakuro puzzles, where a grid may possess a varying number of runs, each with varying run-total and run-length properties.

Table 5.1: Minimum iteration counts using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	16.0	16.0	19.0	4.0	6.0
	4×3	69.0	104.0	65.0	14.0	33.0
	4×4	126.0	278.0	40.0	21.0	22.0
	5×5	733.0	182.0	329.0	113.0	129.0
	6×6	666.0	237.0	66.0	39	48.0
	7×7	762.0	1,613.5	213.0	81.0	96.0
	8×8	840.0	2,625.0	782.0	154.0	1,097.0
	9×9	18,830.0	171,287,499.0	7,948.0	591.0	2,138.0
	10×10	7,499.0	87,543.0	3,454.0	407.0	4,561.0

Table 5.2: Maximum iteration counts using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	93.0	93.0	57.0	13.0	11.0
	4×3	3,901.0	3,034.0	2,121.0	414.0	1,419.0
	4×4	12,523.0	11,697.0	12,309.0	455.0	1,752.0
	5×5	102,164.0	644,063.0	548,422.0	9,668.0	120,110.0
	6×6	478,268.0	5,614,015.0	262,392.0	3,843.0	34,138.0
	7×7	92,772.0	144,733.0	139,150.0	4,820.0	57,623.0
	8×8	52,060.0	944,581.0	34,439.0	2,427.0	28,846.0
	9×9	3,052,061.0	259,592,292.0	17,430,752.0	3,809.0	10,428,114.0
	10×10	38,334.0	3,609,210.0	73,852.0	5,349.0	49,737.0

Table 5.3: Median iteration counts using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	56.0	36.0	23.5	13.0	8.0
	4×3	780.5	654.5	235.0	62.0	125.5
	4×4	1,051.0	1,296.0	650.5	59.0	206.5
	5×5	5,349.0	3,411.0	4,632.0	379.0	4295.0
	6×6	2,295.0	2,801.0	1,186.0	126.0	435.5
	7×7	3,457.0	10,985.5	2,495.0	290.5	1577.5
	8×8	6,273.0	13,983.0	2,972.5	852.5	3140.0
	9×9	75,705.0	10,844,499.0	87,812.0	1,348.0	76215.0
	10×10	24,646.0	274,588.0	26,335.0	1,590.0	11643.0

Table 5.4: Average iteration counts using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	55.3	51.6	28.9	11.2	8.1
	4×3	1,325.6	1,050.4	544.5	122.3	308.6
	4×4	2,717.5	2,958.4	2,156.0	159.4	333.7
	5×5	21,449.5	67,717.2	67,899.1	1,767.7	18,502.1
	6×6	56,185.2	575,000.1	50,298.3	577.8	4,011.7
	7×7	23,014.0	27,742.3	23,651.4	943.8	11,342.4
	8×8	11,372.6	114,235.1	8,029.1	1,030.2	7,151.5
	9×9	649,295.3	64,532,564.8	3,359,825.2	1,712.7	1,313,763.5
	10×10	24,059.7	1,035,084.6	29,362.1	1,849.3	18,256.6

Table 5.5: Median solution times [ms] using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	202.56	137.01	86.01	48.24	30.25
	4×3	2,631.30	2,224.20	811.49	226.05	440.59
	4×4	3,554.68	4,417.27	2,202.80	309.16	712.16
	5×5	17,985.29	11,614.23	15,531.46	1,303.80	14,804.04
	6×6	7,718.42	9,581.84	4,040.54	449.83	1,517.16
	7×7	11,664.28	37,714.61	8,486.47	1,017.25	5,360.00
	8×8	21,698.98	49,660.62	10,190.29	2,954.98	10,905.48
	9×9	269,472.74	38,829,281.19	312,898.56	4,761.14	288,744.51
	10×10	86,355.82	990,972.36	92,699.58	5,599.57	40,919.13

Table 5.6: Average solution times [ms] using each method for each size grouping

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	197.90	190.39	104.98	42.42	30.71
	4×3	4,451.10	3,557.72	1,851.03	428.20	1,058.72
	4×4	9,195.18	10,059.16	7,225.57	558.22	1,136.56
	5×5	72,996.78	235,507.09	246,537.69	6,117.80	65,330.22
	6×6	188,257.92	2,015,169.43	176,924.28	2,003.80	13,903.52
	7×7	79,080.04	96,738.89	83,487.12	3,266.26	40,833.39
	8×8	39,127.80	410,235.16	28,003.24	3,603.94	24,769.24
	9×9	2,297,404.86	233,339,344.65	11,967,608.59	6,045.98	4,863,299.21
	10×10	84,891.62	3,767,504.81	103,490.24	6,547.31	66,583.77

As would be expected, the Value Ordering heuristic worked best on certain puzzles – these being ones in which the first few cells that were considered had high values. The performance of this reverse value ordering relative to the Stack-Based Backtracker Alone approach (forward value ordering) is merely an outcome of the puzzles selected in the test set. Run-Based Cell Ordering was often effective, but seemed less so for larger puzzles; for certain puzzles it performed worse than backtracking alone. The nature of this type of solver means that

cells within different parts of the grid are considered in an order not governed by position alone. This means that the final cell to be considered in a run (whose assignment typically flags more constraint/pruning violations) may be considered at a later stage than if cells of a run are considered sequentially. Such an ordering may generally result in more iterations being required to reach such a stage and hence, more backtracking to correct such erroneous assignments. These "blunt" heuristics are inconsistent in reducing the iteration counts.

In contrast, the projected run pruning performed consistently well, never requiring more iterations than the backtracker alone (as would be expected), and often requiring significantly fewer iterations. Many fruitless paths of the search tree are ignored, meaning fewer iterations and a shorter solution time. A method that combines this rapid and early pruning with a heuristic designed to favourably order assignment is desired. The Decisive Value Ordering approach, when used to solve puzzles with smaller puzzle grids, appears promising. It frequently required only slightly more or fewer iterations, with the overall time taken reasonably consistent. However, for larger grid sizes, the iteration counts tend to increase when compared to the stack-based backtracker alone, possibly due to the uncertainty relating to cells whose solution contains a central value within the valid range. This raises suspicions that the approach could have a vastly detrimental effect on the iteration counts of larger, specific, yet untested puzzle grids. This behaviour may be attributed to the fact that larger grids contain more runs that can be considered long; cells within such runs may need to contain a low value despite being a member of a run with a high run-total or vice-versa. The average for such cells would then suggest that the solver uses an incorrect value order, hence increasing the iteration count. Projected Run Pruning seems to be the most effective approach encountered so far.

Table 5.7: Average time [ms] per iteration

		Approach Used				
		Stack-Based Backtracker Alone	Run-Based Cell Ordering	Value Ordering	Projected Run Pruning (P.R.P.)	Decisive Value Ordering
Grid Size	2×2	3.614024	3.763004	3.644427	3.794735	3.800041
	4×3	3.428015	3.465358	3.465701	3.596174	3.524835
	4×4	3.415889	3.421213	3.440542	3.600350	3.548107
	5×5	3.440743	3.443319	3.427125	3.473198	3.480415
	6×6	3.361836	3.457106	3.486916	3.560521	3.510022
	7×7	3.401571	3.454824	3.475912	3.527476	3.551572
	8×8	3.427775	3.578533	3.462060	3.510232	3.477818
	9×9	3.554802	3.614345	3.542523	3.532072	3.677623
	10×10	3.526051	3.616034	3.512137	3.537155	3.592607

Puzzles of small size generally solve quite rapidly, but the processing overhead of implementing the heuristics and pruning methods is of interest here. Table 5.7 shows the average time taken per iteration, measured in milliseconds, for the puzzles within the same test set defined above. The average time per iteration is generally slightly higher for smaller puzzles due to the processing overhead and time required for puzzle initialisation. It also seems reasonably clear that the processing overheads of run-based cell ordering, which requires additional pre-processing (specifically the "scoring" of each cell and the creation and indexing of an array) are not especially significant when compared to the other solvers.

It is difficult to ascertain other trends in relation to grid size and speed. All of these methods for the automated solution of Kakuro puzzles, on average, take 3.5 milliseconds to perform an iteration, equivalent to approximately 0.28 iterations per millisecond. They essentially all perform the same task; while a solution is yet to be found, cells are visited in a particular order and filled with values in some particular order. This loop continues until a solution is found. Pruning techniques further reduce the number of iterations required (and hence the solution time) but approaches based on this backtracking algorithm still take approximately the same amount of time to perform a single such iteration. While the basic approach of depth-first search is promising, the solution time will become prohibitively large for very large puzzle sizes. A new, faster approach is desired which, as well as keeping the iteration count to a minimum, can perform a larger number of iterations in a given unit time.

5.4 Recursive Methods

In previous approaches, while empty cells existed, the algorithm attempted to assign a valid value to each and every white cell within the puzzle grid, such that no constraint was violated.

This section describes a recursive approach that again employs a form of backtracking algorithm. This approach uses a depth-first examination of the search space, with suitable heuristics to guide the backtracker and effective pruning conditions to reduce the search space size. However, the use of recursion means that the algorithm now attempts to assign a valid value to one white cell only; an apparently successful assignment triggers a recursive call to itself, with the current puzzle state ("Current_State") being passed as a parameter. An unsuccessful assignment returns "false". If and when a solution is found, "true" is returned to the function that made the initial call and the solution is output. The use of recursion therefore eliminates the requirement of a stack. Following the successful Projected Run Pruning (P.R.P.) modification to the Stack-Based Backtracking Algorithms of the previous section, this pruning will also be added to the initial recursive algorithm and to all subsequent modifications (documented in Section 5.5).

Algorithm 5.2: Recursive Backtracking Algorithm: Main()

Initialise global Iteration_Count, Puzzle_Run_Cells, Puzzle_Runtotals and Solution_Stack.
Current_State becomes the initial_state.
Current_Cell is set to be the first available white cell.
if Solve(Current_State, Current_Cell) is TRUE **then**
  Print Solution-Stack.
else
  Print "No Solutions"
end if

Algorithm 5.3: Recursive Backtracking Algorithm: Solve(Current_State, Current_Cell)

```
for Current_Value from 1 to 9 do
  Increment Iteration_Count.
  Determine runs in which Current_Cell resides, and corresponding run-totals.
  Place Current_Value into Current_Cell within Current_State.
  Check resulting Current_State for puzzle violations.
  if [no duplicates in runs] and ([run-total(s) not exceeded] or [run(s) completed correctly]) then
    if No White Cells remain then
      Add Current_State to Solution_Stack
      Return TRUE.
    else
      Current_Cell becomes next available white cell.
      if Solve(Current_State, Current_Cell) is TRUE then
        Return TRUE.
      end if
    end if
  else
    Return FALSE.
  end if
end for
```

In this implementation of a recursive, depth-first approach with pruning, Algorithm 5.2 calls the Boolean "Solve" function (Algorithm 5.3), passing as parameters the initial (empty) grid and a reference to the (first) white cell to be considered. This call is the *initial* call. Algorithm 5.3 then iteratively attempts to assign a value to the white cell passed as a parameter, beginning with the lowest numerical value. An apparently successful assignment of a value to a cell (one which does not violate puzzle constraints) will result in a recursive call to itself, the boolean Algorithm 5.3, passing as parameters the current partially-filled grid and a reference to the white cell to be considered next. Violations of the puzzle constraints - a duplicate value in a run, an exceeded run-total or an under-target run-total where all possible values have been considered for the final cell of a run - will result in the algorithm returning "False" to the parent. If a solution is reached when the last cell has been considered, the

solution is added to a solution stack and "True" is passed to each parent, up to and including that which made the *initial* call, prompting the solution stack to be output. Otherwise, if the largest possible value has been unsuccessfully attempted in all cells, "False" is passed to each parent, up to and including that which made the *initial* call, prompting a "No Solutions" message. This approach may be adapted to find all solutions to a puzzle grid, which is useful to determine whether a given puzzle is well-formed. In such a case, when a solution is found and added to the solution stack, the algorithm continues, instead of passing "True" to the parent, until the highest possible values have been tried in all cells. This approach eliminates the requirement of a stack to store all partially-filled grids along the current branch of the search space, thus avoiding the processing overheads which can arise from its use. An iteration count is incremented each time an attempt is made to assign a value to a cell, and is used as a measure of algorithm performance in Section 5.4.1 and Section 5.5.

5.4.1 Results of Comparing Non-Recursive and Recursive Techniques

Whichever implementation is used, stack-based or recursive, the algorithm must still attempt to assign the same values to the same cells. Therefore, recursive techniques should make no difference to the number of iterations required before a solution is found, when compared to their non-recursive counterparts. Only the time taken to perform each such iteration should decrease, therefore resulting in a decrease in the overall time taken to find a solution. The table below compares the time taken to find a solution using both the recursive backtracking algorithm and the recursive backtracker with P.R.P with their non-recursive counterparts, for each puzzle in the test set used in Section 5.3. For every grid size, ten puzzles are included in the test set. The algorithms being tested are the Stack-Based Backtracker Alone (Section 5.1), Recursive Backtracker Alone (Section 5.4), Stack-Based Backtracker with P.R.P. (Section 5.2.4) and Recursive Backtracker with P.R.P. (Section 5.5). Again, tests were performed on a Viglen Intel Core 2 Duo processor 2.66GHz, with 2GB RAM. Programs were developed using Java platform 1.5.0_06 within Oracle Jdeveloper 10.1.3.3.0, executed in the J2SE runtime environment. All times are measured in milliseconds. For timings, each puzzle is run ten times in succession and the best time recorded.

Tables 5.8 to 5.11 show the minimum, maximum, median and average solution times for

puzzles within each size grouping, measured in milliseconds.

Table 5.8: Comparative minimum solution times [ms]

		Approach Used			
		Stack-Based Backtracker Alone	Recursive Backtracker Alone	Stack-Based Backtracker with P.R.P.	Recursive Backtracker with P.R.P.
Grid Size	2×2	61.20	0.96	15.41	0.21
	4×3	248.00	1.22	53.40	0.55
	4×4	447.65	1.51	79.35	0.72
	5×5	2,503.19	7.67	400.26	3.61
	6×6	2,252.26	5.85	146.85	1.51
	7×7	2,610.66	8.48	301.90	3.08
	8×8	2,882.59	13.43	557.91	8.05
	9×9	65,785.84	216.57	2,106.26	28.21
	10×10	25,932.69	104.95	1,455.90	24.38

Table 5.9: Comparative maximum solution times [ms]

		Approach Used			
		Stack-Based Backtracker Alone	Recursive Backtracker Alone	Stack-Based Backtracker with P.R.P.	Recursive Backtracker with P.R.P.
Grid Size	2×2	326.24	2.00	55.45	0.53
	4×3	13,055.94	21.30	1,418.50	9.35
	4×4	42,289.61	105.98	1,577.67	10.64
	5×5	345,202.66	911.49	33,612.68	150.68
	6×6	1,601,888.96	3,454.16	13,295.45	118.48
	7×7	318,667.91	899.91	16,667.93	180.2
	8×8	179,424.10	544.27	8,484.68	100.14
	9×9	10,725,848.92	33,417.12	13,547.10	163.73
	10×10	135,016.60	653.00	19,074.03	290.52

Table 5.10: Comparative median solution times [ms]

		Approach Used			
		Stack-Based Backtracker Alone	Recursive Backtracker Alone	Stack-Based Backtracker with P.R.P.	Recursive Back-tracker with P.R.P.
Grid Size	2×2	202.56	1.16	48.24	0.31
	4×3	2,631.30	7.51	226.05	1.92
	4×4	5,554.68	7.39	309.16	2.09
	5×5	17,985.29	41.41	1,303.80	10.12
	6×6	7,718.42	18.22	449.83	3.87
	7×7	11,664.28	29.04	1,017.25	10.63
	8×8	21,698.98	72.76	2,954.98	36.46
	9×9	269,472.74	997.16	4,761.14	66.98
	10×10	86,355.82	352.61	5,599.57	91.35

Table 5.11: Comparative average solution times [ms] for recursive methods

		Approach Used			
		Stack-Based Backtracker Alone	Recursive Backtracker Alone	Stack-Based Backtracker with P.R.P.	Recursive Back-tracker with P.R.P.
Grid Size	2×2	197.90	1.22	42.42	0.34
	4×3	4,454.10	8.80	428.20	2.90
	4×4	9,195.18	21.06	558.22	3.86
	5×5	72.996.78	177.56	6,117.80	47.06
	6×6	188,257.92	407.54	2,003.80	17.59
	7×7	79.080.04	243.23	3,266.26	33.90
	8×8	39,127.80	131.70	3,603.94	42.87
	9×9	2,297,404.86	7,963.83	6,045.98	82.80
	10×10	84,891.62	371.03	6,547.31	106.57

The recursive implementation shows a phenomenal improvement to the solution time of all puzzles when compared to the non-recursive, stack-based implementations. The iteration count for each puzzle was unchanged when compared to their non-recursive counterparts. Using the most successful approach encountered so far, recursive projected run pruning,

most grids were solved in less than a second. In comparison, the constraint-based approach of Section 4.4.1, implemented using XPressMP, solved all puzzles with grid sizes between 2×2 and 14×14 and additionally a 29×29 grid in less than a second. These results bear close comparison with the results for the Recursive Backtracker with P.R.P., although the maximum grid size for this approach was 10×10. XpressMP is a commercial application that employs highly optimised methods, and so the author believes the results of this section, using bespoke software that may benefit from further optimisation, compare fairly favourably.

Table 5.12 shows the average time (in milliseconds) taken to perform each iteration within a size grouping by each approach.

Table 5.12: Average time (ms) per iteration

		Approach Used			
		Stack-Based Backtracker Alone	Recursive Backtracker Alone	Stack-Based Backtracker with P.R.P.	Recursive Backtracker with P.R.P.
Grid Size	2×2	3.614023	0.021251	3.794735	0.030157
	4×3	3.428015	0.008377	3.596174	0.026254
	4×4	3.415889	0.008037	3.600350	0.026336
	5×5	3.440743	0.008152	3.473198	0.027752
	6×6	3.361836	0.008698	3.560521	0.031432
	7×7	3.401571	0.009944	3.527476	0.036202
	8×8	3.427775	0.012406	3.530232	0.042879
	9×9	3.554802	0.012704	3.532072	0.049159
	10×10	3.526051	0.014988	3.537155	0.058361

The recursive implementation clearly has a dramatic effect on the overall speed of a solving algorithm compared to the stack-based, iterative implementation. This is despite the opposing viewpoint that recursion incurs a substantial overhead [37], meaning that iterative solutions are usually more efficient than recursive solutions as they do not incur the overhead of the multiple method calls [44] and push states onto the runtime call stack, using both time and memory [17]. Kakuro is a problem with an inherent, natural recursive structure; each cell can be "solved" individually through a single, legal value placement. An

iterative approach to such a problem would require the use of an explicit, user-implemented stack, previously implemented using an "arraylist" within programming language Java. The author acknowledges that the previous stack-based (iterative) approach could have been implemented more efficiently by using a minimal data structure for handling data within the stack. In the implementation previously used, the stack holds puzzle states so was implemented as an "arraylist" of arrays, which can become very large. Also, "arraylists" have additional overheads due to range checks and the internal storage mechanism "under the hood" within Java [57].

However, in relation to this implementation of an automated approach to the solution of Kakuro puzzles, recursion has been shown to make a vastly beneficial difference to solution times. Whereas previous approaches performed less than half of one iteration per millisecond, recursive approaches have now been shown to be capable of performing up to four hundred times faster.

The stack-based, non-recursive approaches performed at a very similar average time per iteration (Table 5.7). This made it difficult to draw any assumptions or conclusions regarding the trends related to speed and grid size. However, it now seems far clearer that for the faster, recursive approaches, speed seems to decrease as the grid size becomes larger, following an initial rise between the 2×2 and 3×3 grids investigated. Smaller puzzle grids are typically solved using a very small number of iterations by the recursive approaches, so initial overheads (reading the puzzle information from files and initialising the variables within the algorithm) take a comparatively larger percentage of the total solution time and hence seems to negate some of the speed benefits, causing the small rise. This behaviour also occurs when solving some specific larger grids where the puzzle is solved using an extraordinarily small number of iterations. For example, the average time taken to perform an iteration within a 4×4 grid is 37.970974 milliseconds, but one particular puzzle, solved using only 21 iterations could perform an iteration in an average of only 29.1652 milliseconds.

As the grid sizes further increase, becoming more typical of standard, published puzzles, the average time taken to complete each iteration increases. Larger puzzle grids contain a larger number of longer runs. This adds a larger time overhead for the trivial checks (whether the placement of a value into a cell violated the non-duplication and/or run-total constraints of the puzzle). A similar overhead exists for the additional pruning checks (whether a run can

possibly be completed given the current assignment of values to the run in question and the automatic insertion of a required value if only a single cell remains empty within such run).

It has been emphasised in Section 5.3 that projected run pruning is effective for the non-recursive, stack-based algorithms. The same benefits are seen for the current recursive counterparts in Tables 5.8 to 5.11. However, on inspection of Table 5.12, a recursive solver without pruning would seem far more desirable due to its greater speed, compared to that of the recursive solver with Projected Run Pruning. Such inference should, however, also consider the information of Table 5.11; despite the relatively slower speed in terms of the number of iterations performed per millisecond, the recursive approach with pruning still solves the overall puzzle using far fewer iterations and in a far quicker, more desirable overall time, particularly for larger puzzle grids compared the recursive approach without pruning. This suggests that the additional overheads of the pruning checks, despite appearing to be dramatically detrimental to speed, are outweighed by the often large decrease in the overall number of iterations required, meaning that the overall solution time is still lower despite the decrease in speed. These overheads are therefore considered worthwhile and the most effective algorithm for the automation of Kakuro puzzle solution is, so far, the recursive backtracking algorithm of Algorithms 5.2 and 5.3 with projected run pruning. In full, Projected Run Pruning performs the following checks (as in Section 5.2.4):

- Consideration of whether a run can possibly be completed to meet its corresponding run-total, given values already assigned to other cells in the run;
- If only one cell remains unfilled within a run, a validity check is performed to calculate the difference between the run-totals of both runs and the current totals. If these differences cannot be met without assigning a value to the remaining cell that is already present in either the horizontal or vertical run in which the cell resides, (hence causing a duplication violation), then this branch of the search space is pruned. Otherwise, the *required* value is added to the cell;
- The remaining value must be no larger than the smallest remainder in run-total from either of the two runs in which it resides and must be less than or equal to nine.

The recursive backtracking algorithm (Algorithm 5.3 of Section 5.4) will be further improved in the following section.

5.5 Modifications to the Recursive Solver

In this section, modifications to the Recursive Algorithm of 5.4 are proposed. The addition of projected run pruning to the recursive algorithm of Algorithms 5.2 and 5.3 is, so far, the best approach presented for the automated solution of puzzle grids within the current test set of puzzles. Therefore, projected run pruning will be present in all further modifications. The results of these modifications are presented and analysed in Section 5.6.

5.5.1 Recursion with Bitmasking

In [54], a recursive method was proposed for the solution of Sudoku puzzles. This approach treats every cell as having a set of nine candidate values and uses a compact binary data representation of a cell such that each of nine binary digits, or *bit* in the structure, represents whether or not a value $1, \ldots, 9$ still remains in the candidate set of the cell. The assignment of a value to the cell results in the corresponding bit in other cells having the same row, column or mini-grid, being set to 0. This process of masking the bits, or *bitmasking*, mimics a human solver's approach to progressively reducing the options for a cell (often executed through the use of "pencil marks"). Here, a similar approach is proposed for Kakuro.

The efficiency of this approach stems from the use of boolean operators NOT, AND, OR and XOR that manipulate the values of each bit within a binary number. It is suggested here that this efficient, compact way of representing puzzle information may decrease solution time.

A sixteen bit binary number is used to represent the status of each white cell in the puzzle (a byte, eight binary bits in length, is unfortunately too short since nine values may be placed in any white cell). The leftmost seven bits are unused and take the value 0. The remaining nine bits represent whether each value can appear in the cell. For example, the rightmost bit represents whether the value 1 can be placed in the current cell and likewise, moving left, other values are represented by remaining bits. If the decimal value held by the data structure for a cell is a power of two, b for example, then a distinct value (from the range $1, \ldots, 9$) $log_2 b + 1$ has been placed into the cell, or is the only value that can legitimately be placed in that cell. The maximum decimal value within a cell is 511, which corresponds to

all nine bits taking the value 1, meaning that any of the nine possible values can potentially be inserted into the cell in question.

This approach sets a bit to 0 if the corresponding value cannot be placed in the given cell without causing a duplicate constraint violation. Unlike other approaches discussed, this approach allows only non-duplicate values to be assigned to a cell, rather than flagging a duplication violation after assignment.

Placing a non-duplicate "Current_Value" into a "Current_Cell", yields two consequences:

- All bits of the data structure corresponding to the "Current_Cell" are set to "zero" except that corresponding to the "Current_Value". This uses the boolean AND function.
- For all cells that share a run with "Current_Cell", bits corresponding to the "Current_Value" are set to "zero". This used the boolean XOR function.

The approach described here assumes that a candidate set for a cell in Kakuro is initially made of the numbers $1, \ldots, 9$ as it would be in Sudoku. It was shown in Section 3.4 that a more restricted candidate set can be derived for a cell in a Kakuro puzzle, by taking advantage of run-total constraints. This concept of a more restricted candidate set is employed in the following three subsections.

5.5.2 Candidate Set-Based Cell Ordering

In Section 3.4, each run within a puzzle was assigned a candidate set; this is a set containing all values that can be used to satisfy the given run-total over the given number of cells. Since each cell is a member of two runs, a cell can only contain the values present in the intersection of the two candidate sets that correspond to the two runs in which it resides. Therefore, this heuristic proposes that by ordering cells for completion such that cells whose intersected candidate sets contain fewest values will be completed earlier than cells whose intersected candidate sets contain more values, less backtracking will generally be required. For example, a run-total of 14 over two cells can be filled using the tuples (5,9), (6,8), (8,6) and (9,5), meaning the corresponding candidate set is {5,6,8,9} for this run-total over two cells. Similarly, an intersecting two-cell run with run-total 6 can be filled using a value from

the candidate set {1,2,4,5}. The intersection of these two candidate sets contains only a unique element, 5, so only this value may be placed in the intersecting cell of the two runs.

Each cell is assigned a score based on the size of the intersection of its candidate sets. Cells having only one value in the intersection of its corresponding candidate sets will be filled first. This partly mimics a human solver's approach to Kakuro. Cells having the same number of values in the intersection of their corresponding candidate sets are ordered according to when they were considered (where cells are examined left to right, row by row in the grid). This ordering takes place in an initialisation stage, and is fixed throughout the search process.

Similarly to its namesake in Section 5.2.1, this particular cell ordering heuristic favours the completion of cells that have fewest valid possibilities in respect to what values can be placed in them. Unlike that namesake, this approach uses information based on single cells rather than assigning cells a score based on whole-run information.

5.5.3 Candidate Set Elimination

This approach also uses the concept of candidate sets that belong to runs within a puzzle (explained in Section 3.4 and above in Section 5.5.2).

Despite the addition of the successful Projected Run Pruning (Section 5.2.4), poor choices of values can still be placed in some cells within a puzzle. As an example, consider a run of 5 cells having the run-total 35. A placement of 5 in the initial cell seems legitimate, because the run-total can be met by placing the required remaining values – 9, 8, 7 and 6 – into the remaining cells. However, if a 5 is not present in the intersection of the candidate sets belonging to this initial cell, considerable processing time would be wasted attempting to fill the remaining cells, until the algorithm was eventually forced to backtrack. By considering whether the "Current_Value" is present in the intersection of the candidate sets belonging to the "Current_Cell", more fruitless branches of the search space can be pruned.

The checks for Projected Run Pruning are extended so that the "Current_Value" is not assigned to the "Current_Cell" unless it is a member of the intersection of the candidate sets of the two runs in which "Current_Cell" resides. A negative result forces the "Current_Value" to be increased by one and a retest is performed. This process is repeated until

an apparently valid value is reached or the "Current_Value" becomes nine. Also, if a cell can only possibly accept one value because the size of the intersection of its candidate sets is one, "Current_Value" now automatically "jumps" to this required value.

It is expected that this approach, in combination with P.R.P. will further decrease the number of iterations required to find a solution and hence, the overall solution time. This should occur as a result of pruning further fruitless branches of the search space and by pruning them potentially earlier. The benefit of reducing the number of iterations is expected to outweigh the cost of any processing overheads.

5.5.4 Hybrid Candidate Set-Based Modification

The modifications of Subsections 5.5.2 and 5.5.3 are combined here; cells that can only possibly accept a lower number of values, based on the size of the intersection of candidate sets are considered before those that can accept a larger number of values. The required value is also automatically inserted into cells that can only possibly accept one value. A "Current_Value" that is not present in the candidate set intersection of the cell in question is automatically increased until an apparently valid value is reached or the "Current_Value" becomes nine.

5.6 Results of Modifying the Recursive Algorithm

The results obtained using the above approaches are compared to results obtained using the best approach so far – the recursive backtracker with projected run pruning – for all puzzle in the set of test puzzles used in Section 5.3. All modifications also include projected run pruning. Tests were performed on a Viglen Intel Core 2 Duo processor 2.66GHz, with 2GB RAM. Programs were developed using Java platform 1.5.0_06 within Oracle Jdeveloper 10.1.3.3.0, executed in the J2SE runtime environment. For timings, each puzzle is run ten times in succession and the best time recorded.

Testing focuses on establishing the relative and general effectiveness of the modifications to the recursive backtracking algorithm of Algorithms 5.2 and 5.3 proposed in Section 5.5: Recursion with P.R.P (Section 5.4), Recursion & Bitmasking (Section 5.5.1), Candidate Set-Based Cell Ordering (Section 5.5.2), Candidate Set Elimination (Section 5.5.3) and the

Hybrid Candidate Set Approach (Section 5.5.4). Tables 5.13 to 5.16 show the minimum, maximum, median and average solution times for each approach within each size grouping, measured in milliseconds. Tables 5.17 and 5.18 show the median and average iteration counts.

Table 5.13: Minimum solution times [ms] for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	0.21	0.74	4.79	4.58	4.71
	4×3	0.55	1.77	5.39	2.18	5.85
	4×4	0.72	2.18	5.95	5.88	6.64
	5×5	3.61	9.47	6.63	7.07	7.01
	6×6	1.51	5.12	7.65	5.40	8.36
	7×7	3.08	10.43	5.81	8.72	7.78
	8×8	8.05	19.09	5.57	12.91	15.77
	9×9	28.21	88.05	16.46	20.28	35.55
	10×10	24.38	71.53	100.70	45.14	60.50

Table 5.14: Maximum solution times [ms] for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	0.53	1.32	5.17	5.16	5.91
	4×3	9.35	25.99	6.90	11.64	7.99
	4×4	10.64	30.49	8.27	12.99	10.98
	5×5	250.68	817.59	260.09	98.03	99.69
	6×6	118.48	322.04	2,055.58	74.25	752.14
	7×7	180.20	547.69	2,366.86	103.07	2,083.99
	8×8	100.14	307.30	278.27	69.99	190.18
	9×9	163.73	656.59	23,915.79	90.33	3,142.69
	10×10	290.52	925.53	653.00	223.35	1,522.16

Table 5.15: Median solution times [ms] for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	0.31	1.30	4.96	4.89	5.31
	4×3	1.92	6.60	5.90	6.91	6.28
	4×4	2.09	6.37	6.67	7.00	7.70
	5×5	10.12	32.28	9.61	12.44	11.49
	6×6	3.87	14.46	9.41	10.78	11.19
	7×7	10.63	33.59	17.75	12.22	14.40
	8×8	36.46	101.37	38.83	34.07	35.57
	9×9	66.98	206.57	296.12	51.29	100.67
	10×10	91.35	253.70	632.71	76.51	518.72

Table 5.16: Average solution times [ms] for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	0.34	1.20	4.97	4.87	5.24
	4×3	2.90	9.23	6.07	7.19	6.66
	4×4	3.86	21.06	6.80	7.47	8.03
	5×5	47.06	143.32	36.50	28.17	20.90
	6×6	17.59	55.82	250.48	17.00	94.21
	7×7	33.90	104.43	272.07	23.67	226.68
	8×8	42.87	128.86	76.65	34.58	51.33
	9×9	82.80	280.57	4,388.28	59.23	472.09
	10×10	106.57	309.55	1,029.62	97.63	554.76

Table 5.17: Median iteration counts for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	13.00	13.00	9.50	8.50	4.00
	4×3	62.00	62.50	28.50	24.50	14.00
	4×4	85.50	86.50	45.00	26.50	18.50
	5×5	379.00	429.00	113.00	224.50	82.00
	6×6	126.00	133.50	109.00	51.50	48.00
	7×7	290.50	339.00	270.00	133.50	167.00
	8×8	852.50	912.00	704.00	573.50	466.50
	9×9	1,348.00	1,456.50	6,037.50	521.00	1,496.50
	10×10	1,590.00	1,751.50	10,111.00	1,010.00	7,711.00

Table 5.18: Average iteration counts for specific puzzles, for each recursive method

		Approach Used				
		Recursion with P.R.P	Recursion & Bit-masking	Candidate Set-Based Cell Ordering	Candidate Set Elimination	Hybrid Candidate Set Approach
Grid Size	2×2	11.20	11.20	9.70	8.40	4.00
	4×3	122.30	124.00	38.30	33.20	16.00
	4×4	159.40	165.50	53.70	59.00	22.70
	5×5	1,767.70	2,042.40	1,073.60	751.50	416.70
	6×6	577.90	647.00	9,047.80	271.70	2,785.60
	7×7	943.80	990.50	7,416.20	407.70	6,108.30
	8×8	1,030.20	1,195.00	1,536.90	571.70	740.50
	9×9	1,712.70	1,926.00	86,172.40	771.00	8,542.10
	10×10	1,849.30	2,260.60	15,738.70	1,364.90	8,036.80

For all modifications, the algorithm speed, measured in iterations per millisecond, generally decreases as the grid size increases. This means that the algorithm works at a faster speed for most of the smaller grid sizes. This may be attributed to the fact that since, as stated previously, larger puzzles typically contain more runs that can be considered "long". For such "long" runs, it is more time consuming to perform the necessary checks that are outlined in the recursive algorithm.

The addition of bitmasking proves disappointing, with increased solution times for all puzzle grids. Checks on the validity of run-sums against the required run-total, such as "over", "under" and "impossible", cannot be performed until a value has been placed in a cell. The restriction that only non-duplicate values are considered for placement in to a cell can delay such checks, meaning that fruitless branches may be pruned less rapidly (leading to a slightly increased iteration count). Results suggest that even when fewer iterations are required, any benefits that are gained from the use of recursion, binary functions and by the more compact representation of cell contents are lost by the large overheads, and the subsequent decrease in speed. This decrease in speed is likely to be caused by the conversion between decimal and binary numbers; the run-totals constraints are calculated in decimal form while the non-duplication of digits constraint requires binary form. It has been reported that Sudoku solvers benefit from a bitmasking approach [54]. While Sudoku problems and Kakuro problems are related, the former is concerned only with the non-duplication constraint while the latter is the interaction of non-duplication constraints with run-total constraints over the same variables. Therefore, bitmasking may be far more beneficial for Sudoku puzzles, or similar puzzles that contain non-duplication constraints alone.

Candidate Set-Based Cell Ordering, based on the size of the intersection of the two candidate sets belonging to a run, also proved disappointing, despite yielding a promising drop in the iteration count for smaller grids. For puzzle grids of size 5×5 and larger, the iteration count actually increases. Given that the solution time also increases as a result of both this iteration count increase and the slower speed of the algorithm (due to the initialisation and intersection of candidate sets for each cell and the initial scoring and re-ordering of all cells within the grid), this approach fails to improve automated efficiency. Cells are filled in an order governed by the number of values in the intersection of its candidate sets. As with the Run-Based Cell-Ordering (Section 5.2.1), cells within different parts of the grid are considered in an order not governed by position alone. The final cell to be considered in a run (whose assignment typically flags more constraint/pruning violations) may be considered at a later stage than if cells of a run are considered sequentially. Such an ordering may generally result in more iterations being required to reach such a stage and hence, more backtracking to correct such erroneous assignments. The hybrid approach showed similar results; whatever beneficial or detrimental difference was made by the Candidate Set-Based Cell Ordering, such a result was only slightly improved by the additional Candidate Set

Elimination.

Candidate Set Elimination, which introduced more pruning based on the candidate sets that belong to each and every run within the grid, shows promise. As expected, the iteration count always decreases in comparison to the recursive solver with P.R.P.; values that can never be assigned to a cell are avoided by examining the contents of the intersected candidate sets that belong to the two runs in which the cell resides. With few exceptions, the average solution time for each grid size group also decreases; however they do not seem to decrease in a manner proportional to the decrease in the iteration count. The median and average solution times actually increase for smaller puzzle grids, despite a decrease in the iteration counts. This suggests that the processing overheads of the Candidate Set Elimination approach have an undesirable effect on the solution times, negating some or all of the benefits of the iteration count decrease, particularly for smaller grid sizes. Ultimately, only the overall time taken to obtain a solution (and whether it was able to arrive at a solution at all) is of importance. More analysis is therefore required to determine whether any decrease in iteration count is warranted by the possible slower nature of this alternative solver (particularly for larger puzzle grids which are more typical of those published), when compared to the best solver so far, Recursion with Projected Run Pruning.

5.6.1 Expanding the Test Set

While few puzzles of small sizes are available, a larger number of published puzzles exist for a more "standard" challenge. For an extended test set of puzzles (200 puzzles of grid size 9×9, 50 of each other grid size up to 10×10), tests are performed on the most promising methods: Recursion with Projected Run Pruning and also with the addition of Candidate Set Elimination. This testing attempts to determine whether the latter approach is more efficient with respect to overall solution time *and* iteration count. Initial testing in Section 5.6 suggests that the latter approach successfully decreases the number of iterations required to reach a solution to all test grids. However, whether the corresponding decrease in speed will negate this iteration count decrease is now under investigation.

With the exception of the 2×2 grids, puzzles have varying difficulty ratings; they have been taken from varying puzzle sources and so their ratings cannot be assumed to be fully

consistent. Typically, puzzles with "easy" or "medium" difficulty ratings contain runs that span fewer cells and, in theory, are easier to complete (Section 3.1.1). All puzzles are well formed, and so possess a single, unique solution. Tables 5.19 to 5.21 show results of the former approach while Tables 5.22 to 5.24 show results of the latter approach when used to solve puzzles in this extended test set.

Table 5.19: Results of Recursion with P.R.P for 10×10, 9×9 and 8×8 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
$\overline{10 \times 10}$	Easy	24.38	1,559.56	100.69	288.15	407.00	29,224.00	1,930.00	5,539.94	18.07
	Medium	11.33	730.05	258.32	308.45	150.00	14,874.00	4,946.00	5,989,53	18.18
	Hard	96.09	6,110.72	618.41	1,357.56	1,625.00	124,337.00	10,608.00	25,115.75	17.74
	Overall	11.33	6,110.72	274.42	637.27	150.00	124,337.00	5,147.50	11,957.06	**18.01**
$\overline{9 \times 9}$	Easy	17.13	16,682.98	208.67	871.50	331.00	330,231.00	4,509.50	17,182.00	20.53
	Medium	80.89	78,827.86	1,150.31	4,112.88	1,594.00	1,615,864.00	23,662.50	85,391.48	20.80
	Hard	40.53	324,183.89	7,965.33	32,407.34	841.00	6,517,160.00	171,486.50	677,380.02	20.82
	Overall	17.13	324,183.89	848.86	11,522.78	331.00	6,517,160.00	17,722.00	240,259.95	**20.72**
$\overline{8 \times 8}$	Easy	7.31	100.14	25.58	38.77	143.00	2,427.00	611.00	886.80	22.30
	Medium	15.18	4,625.14	51.01	380.55	366.00	102,800.00	1,362.00	8,742.82	24.20
	Hard	18.28	2,572.80	88.55	377.63	422.00	60,300.00	1,880.50	9,128.61	24.14
	Overall	7.31	4,625.14	52.30	276.97	143.00	102,800.00	1,281.50	6,524.90	**23.61**

Table 5.20: Results of Recursion with P.R.P for 7×7, 6×6 and 5×5 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
7×7	Easy	2.71	21.30	6.30	7.34	55.00	649.00	155.50	199.89	26.31
	Medium	2.62	54.18	10.41	18.92	60.00	1,597.00	276.00	548.75	27.22
	Hard	8.27	161.24	11.41	37.31	227.00	4,433.00	323.00	1,046.71	28.36
	Very Hard	6.24	180.20	39.21	59.36	140.00	4,820.00	1,085.00	1.656.00	27.57
	Super Hard	7.97	147,211.10	75.30	24,594.85	202.00	4,384,770.00	2,167.00	732,529.83	28.96
	Overall	2.62	147,211.10	9.37	2,972.10	55.00	4,384,770.00	257.00	88,485.62	**27.31**
6×6	Easy	1.51	10.54	2.43	3.58	38.00	402.00	53.00	110.15	28.12
	Medium	1.82	17.45	4.25	6.06	37.00	628.00	123.00	198.78	30.97
	Hard	8.75	4,053.05	185.09	559.25	285.00	140,314.00	6,727.00	19,548.00	33.69
	Overall	1.51	4,053.05	8.37	215.63	37.00	140,314.00	268.00	7,528.44	**31.26**
5×5	Easy	1.54	31.38	4.42	7.35	30.00	1,218.00	135.00	257.18	32.58
	Medium	1.70	259.25	36.70	50.56	58.00	13,198.00	1,289.00	2,190.81	38.41
	Hard	2.55	2,649.17	23.60	201.57	77.00	95,916.00	1,023.00	7,360.53	36.33
	Overall	1.54	2,649.17	11.73	87.21	30.00	95,916.00	415.50	3,291.08	**35.72**

Table 5.21: Results of Recursion with P.R.P for 4×4, 4×3 and 2×2 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
$\underline{4 \times 4}$	Easy	0.64	10.14	1.29	2.24	17.00	429.00	39.00	87.31	34.99
	Medium	0.68	21.35	3.59	5.24	18.00	923.00	149.50	220.56	40.91
	Hard	0.73	40.89	4.21	9.89	21.00	1,930.00	174.50	446.70	40.36
	Very Hard	2.35	43.86	6.50	12.00	84.00	1,927.00	269.50	519.17	41.49
	Super Hard	3.36	67.24	9.33	21.79	141.00	2,782.00	432.00	938.86	44.53
	Overall	0.64	67.24	3.36	8.06	17.00	2,782.00	141.00	346.91	**39.61**
$\underline{4 \times 3}$	Easy	0.54	3.65	0.75	1.12	14.00	192.00	29.00	43.82	36.79
	Medium	0.42	5.59	1.46	1.87	10.00	257.00	55.00	79.29	38.92
	Hard	0.58	9.35	2.06	2.85	16.00	414.00	93.50	122.13	40.81
	Overall	0.42	9.35	1.25	1.93	10.00	414.00	52.50	80.94	**38.80**
$\underline{2 \times 2}$	Overall	0.18	0.66	0.27	0.28	4.00	13.00	12.00	8.54	**30.93**

Table 5.22: Results of Candidate Set Elimination for 10×10, 9×9 and 8×8 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
10×10	Easy	33.04	1,084.21	85.00	189.16	247.00	17,425.00	1,451.00	3,002.47	14.03
	Medium	17.97	631.00	211.80	244.54	91.00	12,598.00	3,135.00	4,098.12	14.91
	Hard	85.66	3,001.37	429.42	914.52	1,113.00	52,488.00	6,914.50	15,074.13	15.58
	Overall	17.97	3,001.37	213.84	440.10	91.00	52,488.00	3,250.00	7,237.92	**14.83**
9×9	Easy	21.09	5,979.21	116.28	416.53	199.00	104,026.00	1,900.50	7,406.10	16.32
	Medium	28.52	9,890.52	527.87	1,323.89	404.00	158,264.00	9,361.50	23,408.22	17.64
	Hard	40.17	64,162.80	1,750.52	9,517.63	523.00	1,356,402.00	33,415.50	180,386.45	18.32
	Overall	21.09	64,162.80	438.88	3,482.86	199.00	1,356,402.00	8,160.50	65,350.58	**17.30**
8×8	Easy	12.49	68.30	28.05	30.77	101.00	1,093.00	451.00	474.20	13.81
	Medium	14.58	1,228.39	49.11	144.24	152.00	23,766.00	1,062.00	2,859.18	18.28
	Hard	20.01	2,265.50	57.63	292.01	273.00	51,876.00	1,205.50	6,409.17	19.26
	Overall	12.49	2,265.50	47.85	163.40	101.00	51,876.00	859.00	3,421.68	**17.29**

Table 5.23: Results of Candidate Set Elimination for 7×7, 6×6 and 5×5 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
7×7	Easy	7.29	19.36	9.76	10.14	31.00	370.00	75.00	95.39	8.45
	Medium	7.68	28.18	12.31	13.94	36.00	570.00	122.50	189.08	11.59
	Hard	8.36	100.98	12.26	23.84	57.00	2,316.00	135.00	426.29	11.70
	Very Hard	11.07	103.25	14.28	30.31	102.00	2,387.00	218.00	577.00	15.01
	Super Hard	11.20	91,794.53	28.84	15,330.31	112.00	2,694,490.00	552.50	449,729.50	19.76
	Overall	7.29	91,794.53	11.39	1,854.21	31.00	2,694,490.00	115.50	54,187.72	**11.94**
6×6	Easy	6.38	12.66	7.23	8.01	21.00	215.00	44.00	62.54	6.95
	Medium	6.65	17.21	7.89	9.33	25.00	359.00	62.00	104.61	9.58
	Hard	10.31	2,179.78	73.88	260.38	131.00	72,375.00	1,939.00	8,241.53	25.99
	Overall	6.38	2,179.78	9.92	104.39	21.00	72,375.00	128.00	3,185.70	**15.13**
5×5	Easy	6.26	32.79	8.19	10.83	27.00	952.00	74.00	176.59	12.08
	Medium	6.37	219.05	19.44	37.07	31.00	9,831.00	492.00	1,239.31	22.51
	Hard	6.68	437.88	16.90	53.91	43.00	13,521.00	360.00	1,546.88	20.05
	Overall	6.26	437.88	14.13	33.88	27.00	13,521.00	296.00	982.56	**18.13**

Table 5.24: Results of Candidate Set Elimination for 4×4, 4×3 and 2×2 puzzles

	Difficulty	Min (best) solution time	Max (best) solution time	Median (best) solution time	Average (best) solution time	Min iteration count	Max iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
4×4	Easy	5.35	12.54	5.97	6.51	11.00	78.00	23.00	27.69	4.38
	Medium	5.46	11.11	6.18	6.80	12.00	222.00	45.00	67.56	8.84
	Hard	5.50	13.20	6.13	7.01	16.00	310.00	39.50	73.30	8.60
	Very Hard	6.60	21.67	8.27	10.71	52.00	615.00	112.50	210.50	16.26
	Super Hard	7.17	37.36	10.24	15.46	67.00	1,332.00	122.00	412.00	19.89
	Overall	5.35	37.36	6.49	8.28	11.00	1,332.00	42.00	116.44	**9.71**
4×3	Easy	5.11	6.72	5.42	5.56	8.00	59.00	13.00	16.94	2.97
	Medium	5.13	8.05	5.60	5.84	8.00	115.00	26.00	32.35	5.10
	Hard	5.32	7.03	6.10	6.06	13.00	80.00	44.50	44.19	7.09
	Overall	5.11	8.05	5.65	5.81	8.00	115.00	20.50	30.90	**5.01**
2×2	Overall	4.65	5.27	4.77	4.79	4.00	13.00	4.00	5.84	**1.22**

Despite appearing detrimental to the times taken to solve puzzles with smaller grids, Candidate Set Elimination is deemed the most successful approach. When compared to Recursive Backtracker with P.R.P., Candidate Set Elimination successfully decreased both the average and median iteration count for all size groupings of puzzles. The average algorithm speed of Recursive Backtracker with P.R.P., specifically the average number of iterations that can be performed in a millisecond, shows a clear decreasing trend as grid size increased. This trend is with the exception of 2×2 grids where puzzle initialisation occupies a larger percentage of the total solution time. The trend is not evident for Candidate Set Elimination, possibly due to success of the new pruning methods reducing the numbers of iterations now required for specific puzzles, especially for easier difficulty ratings. This pruning causes large differences and fluctuations in the speed at which specific puzzles are solved within each size grading, therefore affecting the median and average solution times and speeds. This subsequent decrease in speed for larger puzzles suggests that solution time is likely to slow dramatically for very large grid sizes that might be required for mappings to real-world applications.

As suggested in Section 5.6, the corresponding decrease in the median and average solution times is indeed unproportional to the decrease in the iteration count. However, the processing cost of puzzle initialisation is less evident when larger puzzle grids are solved, which typically require a higher iteration count for their solution. When solving smaller puzzle grids, the median and average solution times actually increase despite the large decrease in the number of iterations required to find a solution. The overheads of generating and intersecting two candidate sets for every cell is clearly more evident in such smaller grids because they are more likely to require a small number of iterations for their solution to be found. Beneficial effects of the pruning associated with these candidate sets would therefore be somewhat negated. Table 5.25 shows how the percentage by which both the median and average iteration counts and solution times for puzzles within this extended test set changed as a result of adding the pruning based on Candidate Set Elimination.

Table 5.25: Changes as a result of Candidate Set Elimination (%) when compared to Recursion with P.R.P.

	Iteration Count		Solution Time	
	Median	Average	Median	Average
10×10	-37%	-39%	-22%	-31%
9×9	-54%	-73%	-48%	-70%
8×8	-33%	-48%	-9%	-41%
7×7	-55%	-39%	+20%	-38%
6×6	-52%	-58%	+18%	-53%
5×5	-29%	-70%	+15%	-61%
4×4	-70%	-66%	+13%	+3%
4×3	-61%	-62%	+352%	+201%
2×2	-67%	-32%	+1,666%	+1,610%

Therefore, in conclusion, Candidate Set Elimination shows great promise for the solution of larger grid sizes that are most typical of published puzzles. Puzzles with these grid sizes typically require a larger number of iterations for their solution, meaning that the benefits of the pruning associated with the candidate sets are more evident. Ultimately, only the overall time taken to determine a solution (and whether it was able to arrive at a solution at all) is of importance. This thesis aimed to find an approach that showed great promise for puzzles within the test sets, for general use for the automated solution of puzzles that are more typical of those published and also of grids that are very large (given sufficient time). This reduction in solution time may not be sufficient for the solution of puzzles with extremely large grid sizes. The presence of many more runs of long length will reduce the frequency with which candidate set restrictions will make a significant difference. It is, however, concluded that despite an discouraging reduction in the solution time required for smaller puzzle grids, the most effective, promising approach is Candidate Set Elimination.

5.7 Summary

Kakuro puzzles do not yield much problem domain information. However, some domain information has been shown to be useful in reducing the number of iterations required for puzzle solution by an automated solver. The useful information is:

1. Run-total violations,
2. Duplication violations,
3. Candidate sets.

All the heuristics presented have associated processing overheads, the effects of which tend to increase as grid size increases. Benefits gained by a reduction in the iteration count may therefore be negated by overheads.

Speed Increases as Grid Size Decreases

With the exception of the 2×2 grids solved, the average number of iterations that were performed per millisecond generally increased as the puzzle grids decreased in size. This means that the algorithm works at a faster speed for most of the smaller grid sizes. This may be attributed to the fact that since, as stated previously, larger puzzles typically contain more runs that can be considered long. For such long runs, it is more time consuming to perform the necessary checks that are outlined in the recursive algorithm, namely:

- Checking each cell in a run to determine whether a duplicate value is present;
- Checking each cell in a run to determine whether the run is full;
- Visiting each cell to determine the current run-total.

The additional pruning component of the algorithms checks whether it is impossible to complete a given run when there is one cell remaining. To achieve this, information about the current sum of the digits in the run in question and the current total of the overlapping run (also containing the one remaining empty cell under consideration) are gathered. This may be time consuming when both these runs are potentially long, as in typical large puzzles, because there are more cells that need to be checked. The considerable increase in the number of iterations required and the subsequent decrease in speed for larger puzzles suggests that solution time is likely to slow dramatically for very large grid sizes that might be required for mappings to real-world applications.

In addition, easier puzzles appear to suffer a decrease in speed compared to puzzles with a harder difficulty rating within the same size grouping. Easier puzzles are typically solved using fewer iterations meaning that the time taken for puzzle initialisation accounts for a higher percentage of the total solution time. The effects of this "initialisation overhead" is

averaged over fewer iterations in easier puzzles. This trend is far more marked for smaller puzzle grids, where the minimum solution times tend to be exceptionally small. For larger grids, the trend, while still evident, is less marked because even the easiest puzzles often require an adequate number of iterations to compensate for the effects of puzzle initialisation.

2×2 Grids Cause Decrease in Speed

Algorithms performed slightly more slowly when solving the set of puzzles with smallest grid size 2×2, despite the fact that these puzzles contain no long runs and are typically solved using a very small number of iterations by the recursive approaches. The benefits of having only short runs appear to be lost because the puzzle initialisation occupies a larger percentage of the total solution time. This means that the processing overheads have more effect on overall solution time. During this initialisation, run information, run-total information and variable initialisation must occur. There are also many fluctuations within each size grouping, affecting the overall average and median results. These are caused by many individual puzzles within each size group now requiring a very small number of iterations for their solution (due to pruning) and is especially true for the Candidate Set Elimination approach.

Harder Puzzles Require More Iterative Exploration

The median number of iterations required to solve a given size of puzzle grid generally increases as the puzzle difficulty rating increases within each size grouping for both approaches that were tested using the extended test set of Section 5.6.1. Higher difficulty ratings increase the likelihood of long runs, which can be filled in many different, potentially correct ways. Increasing run-length tends to delay the detection of violations until a late stage within the solving process. The algorithm could reach some considerable depth before it were forced to backtrack due to a violation, so a higher iteration count is to be expected. Such ratings, discussed in Section 2.1, are often assigned using information about run-length and relate to how difficult a human solver would find it to obtain the solution to the puzzle. This result may imply that an automated solver always requires more iterations when solving "hard" puzzles compared to easier ones, but this is not always true. Some "hard" puzzles require fewer iterations than some rated "easy"; automated solvers do not employ logical methods of deduction, they merely attempt to place values in cells until a solution is found.

Chapter 6

Conclusion

The general aim of this thesis was to address the lack of literature concerning Kakuro puzzle properties to enable consideration of potential applications of the puzzle, following successful applications of other puzzles, notably Sudoku, to real-world problems. The specific aims were (Section 1.2) to determine bounds on the number of valid grid arrangements; to produce a generating function showing the total number of valid, unordered partitions of run-totals into a given number of cells within a run; to determine candidate sets for valid placements of values within cells based on their location in the puzzle; to enumerate the smallest puzzle grids, and consider the possible extension to larger grids; to compare methods of automating the solution of Kakuro puzzles and to investigate whether puzzle properties may usefully be incorporated into heuristics and pruning conditions. A recursive backtracking algorithm, incorporating candidate sets and pruning conditions, was found to be the most effective automated approach.

An individual puzzle, if well-formed, has only one solution, *i.e.* there is only one valid arrangement of values that can be placed into the cells of the grid such that all constraints are satisfied. General upper bound are presented for the number of valid arrangements of values that can be placed into grids when puzzle constraints are relaxed. Section 3.2 presented a crude upper bound, U_1, on the number of valid arrangements of values (from the standard range $1, \ldots, 9$) within a puzzle grid. This ignored both the run-total constraint and the non-duplication constraint, both fundamental to the rules of Kakuro. For a grid with w white cells, $U_1 \leq 9^w$; any cell can contain any of the nine available values. To ascertain the effect of each fundamental puzzle constraint, Section 3.2.1 introduced only the non-duplication

constraint, that cells within the same horizontal or vertical run must contain distinct values, while still ignoring the run-total constraints. However, two cells placed diagonally adjacent to one another may contain equal values. When cells were considered from left to right and top to bottom, an improved upper bound, U_2, was obtained by assuming that the set of values in cells that were horizontally adjacent to the cell under consideration form a subset of the values placed in vertically adjacent cells, assuming there were more vertically adjacent values. The reverse is true if there are more horizontally adjacent values. For an example puzzle grid (Fig. 1.1 of Section 1.1) having sixteen white cells, $U_1 \leq 9^{16} \approx 1.853 \times 10^{15}$. Only a very small number of these arrangements would actually be valid; many would result in there being duplicated values in runs and many more would also sum to incorrect run-totals. For the same puzzle grid, $U_2 \leq 9^2 8^7 7^3 6^4 = 75,511,665,524,736$. Similarly, a lower bound, L_2, is now given that depends on the lowest possible number of values that a cell can validly accept without contradicting the non-duplication constraint. For the example puzzle grid of Fig. 1.1, $L_2 = 9^2 8^6 7^1 6^4 5^3 = 24,078,974,976,000$. The actual number of valid arrangements is 32,920,069,333,536, determined using Algorithm 3.1, the exhaustive counting program of Section 3.2.1.

Section 3.2.2 considered an alternative upper bound by ignoring the non-duplication constraint while the run-total constraint was reintroduced. Not all cells can accept all nine values, even if such values do not cause a duplication violation. To attempt to improve the upper bound on U_2, each white cell was assigned two scores, one for each run to which it belongs. These scores are based on the run-totals that correspond to runs. If a cell belonged to a run of length one, then the cell received a score of one. Otherwise, the cell received a score that was one less than the lowest run-total to which it corresponds, up to a maximum score of nine. For example, a cell belonging to a run of length three with a run-total of 6 cannot possibly accept the values 6, 7, 8 and 9 so would adopt a score of five in relation to this run. An upper bound, U_3, is therefore the product of the lowest scores associated to each white cell. For the example grid of Fig. 1.1, $U_3 \leq 9^5 5^2 4^1 3^3 2^5 = 5,101,833,600$. This was shown in Section 3.2.2 to generally be better than previous upper bounds (with rare exceptions).

To improve on the upper bound, U_3, the cell scoring concept was modified. Each cell was assigned a candidate set of values, based on the length of the run and the corresponding run-total. A candidate set does not contain values that, despite being lower in value than

the lowest run-total associated to the cell, still cannot be placed into the cell. Since a cell is a member of two runs, the values it can actually accept is the intersection of the two candidate sets that correspond to each of its runs. If each cell was assigned a score which reflected the size of such an intersection, a new upper bound, U_4, was the product of the scores assigned to each and every white cell. For the example grid of Fig. 1.1 in Section 1.1, $U_4 \leq 4^4 3^3 2^7 = 884,736$.

In Section 3.2.3, the actual number of valid arrangements of values that can be placed into a grid, avoiding duplication, was generalised in terms of x, where x was the highest valid value that can be used. Since interest lies in the equality or non-equality of diagonal cells, the problem was divided into a number of cases. The equality or non-equality of an *negative diagonal* pair of cells (and combinations of two or more of such pairs) represented distinct cases, *positive diagonal* pairs (or combinations of two or more of such pairs) represent *sub-cases*. Sub-cases were only taken into consideration if there were white cells in row three or below. Since constraints prohibit only the placement of equal values into horizontal and vertical runs, diagonally adjacent cells may accept equal values. The cases and sub-cases therefore represent whether a given diagonal pair contains equal or distinct values. By firstly investigating cases within a small 2×2 grid and augmentations thereof, the method evolved to tackle the grid of Fig. 1.1; due to the size of the grid and the evident number of cases, the problem was instead split into two disjoint sub-grids. Unfortunately, the problem was still cumbersome so required the use of Gaussian elimination to obtain a final result. In terms of x, the grid of Fig. 1.1 can be filled using the number of valid arrangements shown in Equation 3.25 of Section 3.2.4.2. Therefore, if $x = 9$, the grid of Fig. 1.1 can accept 32,920,069,333,536 valid arrangements of values. This result was computationally confirmed using Algorithm 3.1, the exhaustive counting program of Section 3.2.1.

Although there are only 502 sets of values that can be used to fill runs of all possible run-lengths that sum to all possible run-totals, when the alternative orderings of these sets of values is taken into account, there are $986,400$ different orderings. It is the interlocked nature of these runs, rather like a Crossword puzzle, that governs which of these runs should be used. A generating function was developed in Section 3.3. This generating function provided a means to evaluate the complexity of a given run, based on its associated run-total and length. Run complexities were then used for scoring purposes in a cell ordering approach in Section 5.2.1. Analysis of the coefficients obtained from a series expansion of

the generating function showed the total number of valid, unordered arrangements of values that can be used to satisfy a given run-total over the required number of cells. Since order is important within a run, multiplying the desired coefficient by the run length in question gave the number of ordered compositions for the desired run. This generating function was used to develop a look-up table that has been employed in a heuristic in Section 5.2.1.

Upper bounds calculated using only the run-total constraint are far lower than those calculated using only the non-duplication constraint. The interaction of these two constraints will further decrease the upper bound on the number of valid arrangements of values into a grid, so that the bound becomes the actual number of grid arrangements that exist for a given grid. Again, only those puzzle grids that are well formed, possessing one solution, are usually published. Both constraints are therefore introduced in Section 3.4. As briefly mentioned above, each cell belongs to two runs and each of these runs can be assigned a candidate set. Such candidate sets detail exactly what numbers can be used to satisfy the given run-total over the given number of cells. Therefore, each cell can only possibly accept the values present in the intersection of the two candidate sets that correspond to the runs to which it belongs. Initially for a 2×2 grid in Section 3.4, all arrangements were enumerated. This enumeration process assumed that the values to be placed in the upper, right cell depended on the intersection of the two candidate sets associated with it. Placement of the correct, necessary values in the upper, left cell and the lower, right cell were trivial, leaving the lower, left cell to accept up to seven values. Care was taken to ensure that grids were not multiply counted through reflection and rotation until all arrangements of values within a 2×2 grid had been considered. When reflections and rotations were then taken into account, there were 4,104 valid arrangements of values within a 2×2 grid. This result was computationally confirmed using Algorithm 3.1, the exhaustive counting program of Section 3.2.1. Only a small number of these grid arrangements would be well-formed; those that possess a single solution given the run-totals present. Unfortunately, as grid size increased, complexity increased disproportionally. Larger grids may have corner or central cells removed and, due to their size, the required value to be placed in some cells is no longer trivial because some runs are now longer. Enumerating larger grids in this way therefore remains future work.

Chapter 4 described standard methods for automating the solutions to Kakuro puzzles. Each general approach was described and consideration was then given to how the approach may

be implemented for Kakuro, hence enabling the evaluation of the usefulness of each approach.

Section 4.1 described implementations of exhaustive search (with and without backtracking), for a state-based representation of the problem. Although, in theory, it is possible to solve any state-based problem using exhaustive search approaches, in practice, it is not efficient and highly time consuming. While exhaustive search methods are certainly appropriate for Kakuro puzzles with smaller grids, or those that use a smaller range of values, it would be inadequate for larger puzzle grids without the addition of pruning conditions, associated with backtracking, to reduce the number of states that need to be explored. Since all solutions lie at the deepest level within the search space, an implementation of a depth-first search approach would be efficient as fewer states would have to be stored (at most, the number of states equal to the depth of the tree). Therefore, a depth-first approach, with the pruning conditions of backtracking, was deemed to be promising for the solution to Kakuro puzzles.

Section 4.2 extended the state-based search concept to consider a local search approach. During such an approach, the local neighbourhood of a state is found by applying a valid operators to the current state to derive successor states. The merit (or score) of each successor is evaluated by some objective function, that employs problem domain information, and the best such successor state is then selected for further subsequent expansion. The numeric nature of Kakuro puzzles initially suggested that the puzzle could allow the use of a fairly simple scoring system. However, the amount of problem domain information that related specifically to Kakuro was limited – only how closely the current run sums matched specified run-totals. Therefore, the limited amount of information that could usefully be incorporated into an effective objective function may be detrimental to the effectiveness of such a function. Hence, it is expected that the method would become stuck in plateaus in the search space, where many states have the same assigned score. There were not strong reasons to favour local search approaches when a solution may be found using an exhaustive approach with pruning.

Section 4.3 introduced the use of metaheuristics, including tabu search and genetic algorithms. Metaheuristics are used to solve various computational problems by adding a high-level algorithmic approach that guides existing control strategies and heuristics in a search for feasible solutions. A tabu search approach employs prohibition-based techniques that

complement basic search algorithms. Following an optimal change to the current state, one implementation of the *tabu* aspect may memorise the most recent operation and prevent its re-use for a set period. Genetic algorithms consist of populations of "chromosomes", which represent states of some problem search space. Genes within a given population have evolved from those within an initial population through specific mutation and crossover operations and contain advantageous properties of "parent" genes. Metaheuristic techniques may be susceptible to any failings of the underlying control strategy or heuristic. The objective function would still be used to assign scores to states within the given neighbourhood. As with the local search approach, the amount of problem domain information relating specifically to Kakuro is limited. The likelihood may be that the scores of the solutions in the pool would converge around some local optimum, or plateau, requiring mutation to escape. The effectiveness of a genetic algorithm is likely to depend on whether a crossover point can be chosen such that good solution characteristics of the parent chromosomes are preserved. For Kakuro, the choice of crossover point may be problematic, due to the number of runs that may need to be broken. Metaheuristic approaches often involve high processing overheads. The simpler approach offered by exhaustive search with pruning was preferred.

Section 4.4 introduced a constraint based approach to problems. Constraint based approaches view a given problem as a collection of variables whose values must be assigned such that various stated constraints are satisfied. Such an approach, using binary integer programming (one form of a constraint satisfaction problem), was implemented by the author [14] for Kakuro puzzles, since they possess explicit puzzle constraints. The approach found a solution for a range of puzzle grids ranging up to 29×29 in size in under one second, but due to a lack of computer memory, failed to find a solution for the largest available grids, 35×35 and 39×39, where there are large amounts of variables present. This is a computer limitation rather than a software limitation so it is thought that with the use of a more powerful machine, a solution can be found, given enough resources. The binary integer programming approach outputs only the first solution it arrives at; since well-formed Kakuro puzzles possess only one solution, this would not ordinarily be a problem. However, this method could not therefore be used to verify that a given puzzle does indeed have a unique solution. The preference for a method of automated solution that ensures solution uniqueness, and which is extendible to solutions of puzzles of large sizes, lead to a rejection of constraint satisfaction approaches.

The most promising approach of Chapter 4 was considered to be a depth-first implementation of exhaustive search with the pruning benefits of backtracking. The first implementations employed a *stack-based* approach. All implementations of this type performed at a fairly constant speed of approximately 0.28 iterations per millisecond, so were compared solely by the number of iterations required, as explained in Section 5.1. Projected Run Pruning introduces additional pruning rules to the stack-based backtracking algorithm of Section 5.1. On assigning a value to a cell in a run that still possesses unassigned cells, a calculation is performed to ascertain whether the associated run-total can be met by placing the highest available values. Otherwise, backtracking occurs, and the remaining states in the branch are pruned. In addition, when one cell remains, backtracking also occurs if the remaining cell cannot be satisfied by a non-duplicate value from the standard, valid range. Such Projected Run Pruning successfully reduced the iteration count (as would be expected) and proved most effective in terms of improving solution times when compared to the backtracking algorithm alone. Other modifications (the results of which were given in Section 5.3) implemented were:

1. A cell ordering heuristic (Section 5.2.1) which proved to be unreliable, possibly because cells within different parts of the grid are considered in an order not governed by position alone. This means that runs may possibly be completed later in the search, in turn, taking longer for constraint/pruning violations to be detected.

2. As expected, a value ordering approach (Section 5.2.2), reversing the range of values to be placed in cells from $1, \ldots, 9$ to $9, \ldots, 1$, improved the iteration count for some grids. However, it was detrimental for puzzles in which the solution of the first few cells considered had low values.

3. An alternative value ordering approach (Section 5.2.3) employs a decision made on a cell by cell basis as to whether to use the standard range or its reversed counterpart. Particularly while solving larger grids, cells within some runs may need to contain a high value despite being a member of a run with a low run-total, or vice-versa, which would suggest an ambiguous average for such cells, causing the solving algorithm to use an incorrect value order, possibly increasing the iteration count. There is also uncertainty about what ordering should be used when the average of a cell is central.

The average time per iteration is generally slightly higher for smaller puzzles due to the

processing overhead and time required for puzzle initialisation. It is difficult to ascertain other trends in relation to grid size and speed. All of these methods for the automated solution of Kakuro puzzles, on average, take approximately 0.28 iterations per millisecond. A faster approach was required which was capable of increasing the actual number of iterations performed per unit time. *Recursive* implementations were employed, to take advantage of the inherent, natural recursive structure of Kakuro puzzles; each cell can be "solved" individually through a single, legal value placement.

The recursive implementations, when compared with the non-recursive implementations performed up to four hundred times faster. Whereas approaches of Section 5.1 performed up to 0.295 iterations per millisecond, speeds of up to 122.6 iterations per millisecond were now possible, albeit for the smaller grids. This is despite there being opposing viewpoints that recursion incurs a substantial overhead [37], meaning that iterative solutions are usually more efficient than recursive solutions as they do not incur the overhead of the multiple method calls [44] and push states onto the runtime call stack, using both time and memory [17]. Kakuro is a problem with an inherent, natural recursive structure; each cell can be "solved" individually through a single, legal value placement. A recursive approach may therefore be more efficient, since an iterative approach to such a problem would require the use of an explicit, user-implemented stack, previously implemented using an "arraylist" within programming language Java. However, the author acknowledges that the previous stack-based (iterative) approach could have been implemented more efficiently (Section 5.4.1).

Projected Run Pruning was shown to be a worthwhile addition to the backtracking algorithm of Section 5.2.4, so was added to all recursive modifications. Processing overheads of such pruning could now be seen far more clearly; the additional checks required during each iteration caused an apparently large decrease in speed (shown in Table 5.12). However, Table 5.11 showed that overall solution times were still optimal due to the iteration count decrease despite the corresponding decrease in speed.

Since changes in algorithmic speed, measured in iterations per millisecond, are now far more evident, speed as well as iteration count was examined. The approach that most improved the solution times for grids tested was the incorporation of candidate sets into approaches, which provide a useful method for evaluating which of the standard values can actually be placed into each cell. More pruning techniques were added in Section 5.5.3 which utilised

the fact that despite the current (projected run) pruning in place, not all values could be placed in all cells. If a cell can only accept one value, then it was filled accordingly. Also, if the current value in question is not present in the intersected candidate set of the cell under consideration, the current value is increased until a value is reached that is. At worst, this approach will make no difference to the iteration count for a puzzle. With few exceptions, this modification successfully lowered the average solution time for each grid size group; however they do not seem to decrease in a manner proportional to the decrease in the iteration count, so further subsequent investigation was required. Other modifications (the results of which were given in Section 5.6) implemented were:

1. A bitmasking approach treated each cell as having a set of nine candidate values. A compact binary data representation of each cell is used such that each of nine binary digits, or *bits* in the structure, represented whether or not a value $1, \ldots, 9$ still remained in the candidate set for the specific cell. Results suggest that when a greater number of iterations are required, any benefits gained from the use of recursion, binary functions and by the more compact representation of cell contents are lost by the large overheads caused by the use of both decimal and binary numbers and conversions between such numbers. It is thought that this approach may be far more beneficial for Sudoku puzzles, or similar puzzles that contain non-duplication constraints alone.

2. An alternative cell ordering heuristic was employed in Section 5.2.1. Each cell is a member of two runs and can only contain the values present in the intersection of the two candidate sets (Section 3.4) that correspond to the two runs in which it resides. This heuristic proposes that by ordering cells for completion such that cells whose intersected candidate sets contain fewest values will be completed earlier than cells whose intersected candidate sets contain more values, less backtracking will generally be required. As with the previous cell ordering approach, this modification proved to be unreliable since cells within different parts of the grid are considered in an order not governed by position alone. Again, this means that it may take longer for constraint/pruning violations to be detected, requiring more backtracking to correct such erroneous assignments. Speed also suffered a slight decrease due to the additional overheads of deriving two candidate sets for each cell, finding the intersection and cell re-ordering.

3. A hybrid of candidate set elimination and candidate set based cell ordering (above). This approach showed similar results to the candidate set-based cell ordering approach;

whatever beneficial or detrimental difference was made by the Candidate Set-Based Cell Ordering, such a result was only slightly improved by the additional Candidate Set Elimination.

To ascertain which approach was the most effective automated solving approach within the recursive implementations, the recursive approach with projected run pruning alone, and with the candidate set elimination approach were used to solve a larger test set of puzzles. In particular, it was necessary to determine whether the promising decreases in the average and median iteration counts for the latter approach are accompanied by decreases in the average and median solution times. Recursive pruning with candidate set elimination dramatically decreased both the median and average iteration counts for *all* puzzles within each size group of the expanded test set. However, as was the case when the smaller test set was used, the decrease in the median and average overall solution time was unproportional to the corresponding decrease in the number of iterations required. On several occasions, puzzles took longer to be solved despite a large decrease in the number of iterations required to find their solution. Table 5.25 showed how the percentage by which both the median and average iteration counts and solution times for puzzles within this extended test set changed; solution times for the smallest grids were greatly increased, suggesting the infeasibility of this candidate set elimination approach. However, larger puzzle grids (more typical of those that are published) required less time for their solution and a far lower number of iterations. It is thought that for puzzles that are much larger in size, the benefits of such an approach may not be sufficient for solution in a reasonable length of time. Nevertheless, it is therefore concluded that despite an apparent detrimental effect on the solution times of smaller grids, the most effective approach is the Recursive Backtracking Algorithm with Projected Run Pruning *and* Candidate Set Elimination.

Although the recursive backtracking algorithm incorporating projected run pruning and candidate set elimination improved solution times, these, and all modifications, showed a decrease in speed as the grid sizes increased. This means that the algorithm works at a faster speed for most of the smaller grid sizes, which may be attributed to the fact that larger puzzles typically contain more runs that can be considered "long". For such long runs, it is more time consuming to perform the necessary checks that are outlined in the recursive algorithm. The rate of decrease in speed as grid size increases limits the usefulness of Kakuro in real world applications that require a mapping to very large grids. In partic-

ular, applications in error-correcting codes would require the use of very large grids (in a similar way to a proposed use of Sudoku [48]).

6.1 Future Work

The findings reported in this thesis suggest a number of useful areas of research. All grids of size 2×2 were enumerated in Section 3.4.3. When grid size increases, there are more white cells and the patterns of white cells are no longer necessarily rectangular (since the corner cells may now be black for example). It would be useful to determine whether all grids of a larger, given size could be fully enumerated. Any results could be verified by using Algorithm 3.1, the exhaustive counting program of Section 3.2.1.

Section 4.3 outlined how a genetic algorithm approach may be formulated for Kakuro. The effectiveness of a genetic algorithm is likely to depend on whether a crossover point can be chosen such that good solution characteristics of the parent chromosomes are preserved. It would be beneficial to know whether it is possible, for some puzzles, to choose the crossover point (a single cell) such that a sufficient number of unbroken runs are located either side of the crossover to enable child chromosomes to inherit useful solution characteristics. The general effectiveness of such approach may then be evaluated.

Binary integer programming solved a range of puzzle grids, ranging up to and including 29×29 in size, very quickly. Larger grids could not be solved due to a limitation on the amount of computer memory available. Despite the fact that this method could not be used to verify that a given puzzle does indeed have a unique solution, with more computer memory and resources available, it is thought that this approach could be used to solve puzzles consisting of very large grids.

Bibliography

[1] advance features.co.uk. A sample British Crossword grid. 2007. `http://www.advance-features.co.uk/images/crossword.gif`.

[2] AUS-PC-SOFT. A sample American style Crossword. 2008. `http://www.crauswords.com/`.

[3] R. A. Bailey, P. J. Cameron, and R. Connelly. Sudoku, Gerechte designs, Resolutions, Affine Space, Speads, Reguli and Hamming Codes. *American Mathematical Monthly*, 115 (5):383–404, 2008.

[4] N. Beaumont. A logical branch and bound algorithm. In *Presented at Australian Society for Operations Research Conference on Mathematical Programming*, 1986.

[5] N. Beldiceanu, M. Carlsson, and J. Rampon. Global constraint catalog. Technical Report T2005:08, SICS, May 2005.

[6] M. Bóna. *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory*. World Scientific Publishing Co. Pte. Ltd., 2nd edition, 2006.

[7] M. Cadoli and M. Schaerf. Partial solutions with unique completion. *Lecture Notes in Computer Science*, 4155:101–115, 2006.

[8] T. Cazenave. A search based sudoku solver. Available at: `http://www.ai.univ-paris8.fr/~cazenave/sudoku.pdf`, France, 2006.

[9] A. Chisholm. How to solve Kakuro puzzles. 2005. `http://www.indigopuzzles.com/ipuz/`.

[10] Conceptis Puzzles. *The Big Book of Kakuro Puzzles*. Sterling Publishing Co., 2006.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[12] D. A. Cox and J. B. Little. *Ideals, varieties, and algorithms : an introduction to computational algebra. - 3rd ed.* Springer, 2007.

[13] R. P. Davies, P. A. Roach, and S. Perkins. Automation of the solution of Kakuro puzzles. In M. Bramer, F. Coenen, and M. Petridis, editors, *Research and Development in Intelligent Systems XXV: Proceedings of AI-2008, the Twenty-eighth SGAI International Conference on Artificial Intelligence*. Springer-Verlag, 2008.

[14] R. P. Davies, P. A. Roach, and S. Perkins. Properties of, and solutions to, kakuro and related puzzles. In P. Plassmann and P. A. Roach, editors, *Proceedings of the 3rd Research Student Workshop*, pages 54–58. University of Glamorgan, March 2008.

[15] R. P. Davies, P. A. Roach, and S. Perkins. The use of problem domain information in the automated solution of Kakuro puzzles. *International Journal of Computer Science (IJCS)*, Submited 2009.

[16] I. Dotu, A. del Val, and M. Cebrian. Redundant modeling for the quasigroup completion problem. In R. F, editor, *Lecture Notes in Computer Science*, volume 2833, pages 288–302, Berlin, 2003. Springer-Verlag.

[17] P. Drake. *Data structures and algorithms in Java*. Prentice Hall, 2006.

[18] B. Felgenhauer and F. Jarvis. Mathematics of Sudoku I. *Mathematical Spectrum*, 39:15–22, September 2006.

[19] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1996.

[20] T. Forbes. Quasi-Magic Sudoku puzzles. *M500*, 215:1–10, April 2007.

[21] J. Gago-Vargas, I. H. Hermoso, and J. M. Morales. Sudokus and Grobner bases not only a divertimento. *Lecture Notes in Computer Science*, 4194:155–165, 2006. `http://www.ricam.oeaw.ac.at/Groebner-Bases-Bibliography/gbbib_files/publication_1180.pdf`.

[22] G. Galanti. The history of kakuro. *Conceptis Puzzles*, 2005. http://www.conceptispuzzles.com/articles/kakuro/history.htm.

[23] gameszoo.org. A sample Kakuro puzzle. 2005. `http://www.gameszoo.org/custom/kakuro/example.jpg`.

[24] C. Gomes and D. Shmoys. The promise of LP to boost CP techniques for combinatorial problems. In N. Jussien and F. Laburthe, editors, *Proceedings of the Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 291–305, France, 2002. CPAIOR.

[25] R. P. Grimaldi. *Discrete and Combinatorial Mathematics*. Addison Wesley Longman, Inc, 4 edition, 1999.

[26] P. Grosse. Kakuro trivia. 1996. `http://www.grosse.is-a-geek.com/kaktriv.html`.

[27] S. Gupta. Some results on Su Doku. Available at: `http://theory.tifr.res.in/~sgupta/sudoku/theorems.pdf`, March 2006.

[28] V. Hanssen. Kakuro puzzles. 2009. `http://www.menneske.no/eng/`.

[29] G. Harris. Generation of solution sets for unconstrained Crossword puzzles. In *Proceedings of the 1990 Symposium on Applied Computing*, volume 5-6, pages 214 – 219, April 1990.

[30] B. Hayes. Unwed numbers. *American scientist*, 94(1):12–15, 2006.

[31] A. Heyworth. Gröbner basis theory. 2001. `http://www.cs.le.ac.uk/people/ah83/grobner/`.

[32] S. K. Jones, S. Perkins, and P. A. Roach. Properties, isomorphisms and enumeration of Quasi-Magic Sudoku. *Submitted to Discrete Mathematics*, 2009.

[33] S. K. Jones, P. A. Roach, and S. Perkins. Construction of heuristics for a search-based approach to solving SuDoku. In M. Bramer, F. Coenen, and M. Petridis, editors, *Research and Development in Intelligent Systems XXIV: Proceedings of AI-2007, the Twenty-seventh SGAI International Conference on Artificial Intelligence*, number 3, pages 37–49. Springer-Verlag, 2007.

[34] S. K. Jones, P. A. Roach, and S. Perkins. Properties of Sudoku puzzles. In P. Plassmann, editor, *Proceedings of the 2nd Research Student Workshop*, pages 7–11, November 2007.

[35] kakuro.com. Kakuro information. 2009. `http://www.kakuro.com/howtoplay.php`.

[36] A. Lerner. Solving the Kakuro puzzle. page 1, 2008. `http://www.cs.tau.ac.il/~alan`.

[37] Y. D. Liang. *Introduction to Java Programming: Comprehensive Version*. Prentice Hall, 2008.

[38] M. L. Littman, G. A. Keim, and N. Shazeer. A probabilistic approach to solving Crossword puzzles. *Artificial Intelligence*, 134:23–55, 2002.

[39] G. F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (Fifth Ed)*. Addison Wesley, 2005.

[40] Maplesoft. Maple: Math software for engineers, educators & students (version 12). 2008. `http://www.maplesoft.com/`.

[41] Z. Michalewicz and D. Fogel. *How to Solve It : Modern Heuristics*. Springer-Verlag, Berlin, 2000.

[42] M. Michalowski, C. A. Knoblock, and B. Y. Choueiry. Exploiting problem data to enrich models of constraint problems. `http://www.cse.cuhk.edu.hk/~jlee/cp07Model/pdf/exploiting.pdf`.

[43] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[44] I. Mitrovic. Replace recursion with iteration. `http://www.refactoring.com/catalog/replaceRecursionWithIteration.html`.

[45] S. Mustonen. On certain Cross Sum puzzles. *Internal Report*, 2006. `http://www.survo.fi/papers/puzzles.pdf`.

[46] S. Mustonen. On the swapping method. *Internal Report*, 2007. `http://www.survo.fi/puzzles/swapm.html`.

[47] K. Nabeshima and F. Winkler. *Comprehensive Gröbner Bases in Various Domains*. PhD thesis, Johannes Kepler Universität, 2007. `http://www.risc.uni-linz.ac.at/publications/download/risc_3116/Nabeshima_thesis.pdf`.

[48] L. A. Phillips, S. Perkins, P. A. Roach, and D. H. Smith. Sudoku and error-correcting codes. In P. Roach, editor, *Proceedings of the 4th Research Student Workshop*, 2009.

[49] J. S. Provan. Sudoku: Strategy versus structure. In H. Crapo, G. Gordon, and J. Oxley, editors, *Brylawski Memorial Conference*. University of North Carolina, 2008.

[50] T. D. M. Purdin and G. Harris. A genetic-algorithm approach to solving Crossword puzzles. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 263 – 270, 1993.

[51] M. Qasem. Sat and max-sat for the lay-researcher. Available at: http://users.ecs.soton.ac.uk/mqq06r/sat/, 2009.

[52] V. Rayward-Smith, I. Osman, C. Reeves, and G. Smith, editors. *Modern Heuristic Search Methods*. John Wiley & Sons ltd, 1996.

[53] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, 2 edition, 1991. Singapore.

[54] P. A. Roach, I. J. Grimstead, S. K. Jones, and S. Perkins. A knowledge-rich approach to the rapid enumeration of Quasi-Magic Sudoku search spaces. In J. Filipe, A. Fred, and B. Sharp, editors, *Proceedings of ICAART 2009, the 1st International Conference on Agents and Artificial Intelligence*, pages 246–254, Porto, Portugal, January 2009.

[55] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education International, 2 edition, 2003.

[56] T. Seta. The complexities of puzzles, Cross Sum and their 'Another Solution' problems (ASP). Master's thesis, University of Tokyo, February 2006.

[57] J. Shirazi. The performance of java's lists. page 1, 2001. http://onjava.com/pub/a/onjava/2001/05/30/optimization.html?page=1.

[58] H. Simonis. Sudoku as a constraint problem. In H. B. P. P and S. B, editors, *Proceedings of the Fourth International Workshop*, pages 13–27. CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems, CP, 2005.

[59] H. Simonis. Kakuro as a constraint problem. 2007. http://4c.ucc.ie/~hsimonis/kakuro.pdf.

[60] M. Skinner. Genetic algorithms overview. 2009. http://geneticalgorithms.ai-depot.com/Tutorial/Overview.html.

[61] D. Smith. So you thought Sudoku came from the land of the rising sun. *The Observer*, Sunday May 15th 2005.

[62] E. Soedarmandji and R. J. McEliece. Iterative decoding for Sudoku and Latin square codes. In *Forty-Fifth Annual Allterton Conference*, pages 488–494, 2007.

[63] R. Sweet. A sample of a fully interlocked Crossword puzzle. 2002. http://www.lafn.org/~keglerron/kegler.html.

[64] M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. *ICL Systems Journal*, 12, May 1997.

[65] websudoku.com. Sample Sudoku & Rudoku puzzles. 2005. `www.websudoku.com`.

[66] E. W. Weisstein. Gröbner basis. 2000. `http://mathworld.wolfram.com/GroebnerBasis.html`.

[67] D. B. West. *An Introduction to Graph Theory*. Prentice Hall, 2001.

[68] J. M. Wilson. Crossword compilation using integer programming. *The Computer Journal*, 32:273–275, 1989.

[69] X. Yang. *Cryptic Kakuro and Cross Sums Sudoku*. Exposure Publishing, 2006.

[70] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. In *Proceedings of the National Meeting of the Information Processing Society of Japan*, Japan, 2002. IPSJ.

Appendices

Appendix A

Using the Diagonal Pairs Method for a 5×5 Puzzle Grid

The Diagonal Pairs method of Section 3.2.3 can be used to count how many ways exist to place values from the range $1, \ldots, x$ into each cell without duplication in runs. Originally, the puzzle grid of Fig. 3.9 was considered but discontinued due to the number of negative and positive diagonal pairs, and hence cases and sub-cases present. For ease, cells are labeled with and referred to by a unique letter label. To demonstrate the cumbersome nature of this infeasible method, the following table lists the cases derived before the process was discontinued. Each case would also have a very large number of sub-cases.

$c = a$	$c = b$ and $k = i$	$g = b$ and $k = e$	$g = f$ and $o = j$	$k = b$ and $o = m$
$c = b$	$c = b$ and $k = j$	$g = b$ and $k = f$	$g = f$ and $o = m$	$k = b$ and $o = n$
$d = b$	$c = b$ and $o = f$	$g = b$ and $k = i$	$g = f$ and $o = n$	$k = b$ and $l = f$
$g = a$	$c = b$ and $o = i$	$g = b$ and $k = j$	$g = f$ and $l = f$	$k = b$ and $l = i$
$g = b$	$c = b$ and $o = j$	$g = b$ and $o = f$	$g = f$ and $l = i$	$k = b$ and $l = j$
$g = d$	$c = b$ and $o = m$	$g = b$ and $o = i$	$g = f$ and $l = j$	$k = b$ and $m = j$
$g = e$	$c = b$ and $o = n$	$g = b$ and $o = j$	$g = f$ and $m = j$	$k = b$ and $p = j$
$g = f$	$c = b$ and $l = f$	$g = b$ and $o = m$	$g = f$ and $p = j$	$k = b$ and $p = n$
$h = b$	$c = b$ and $l = i$	$g = b$ and $o = n$	$g = f$ and $p = n$	$k = e$ and $k = i$
$h = e$	$c = b$ and $l = j$	$g = b$ and $l = f$	$h = b$ and $h = f$	$k = e$ and $k = j$
$h = f$	$c = b$ and $m = j$	$g = b$ and $l = i$	$h = b$ and $k = e$	$k = e$ and $o = f$
$k = b$	$c = b$ and $p = j$	$g = b$ and $l = j$	$h = b$ and $k = f$	$k = e$ and $o = i$
$k = e$	$c = b$ and $p = n$	$g = b$ and $m = j$	$h = b$ and $k = i$	$k = e$ and $o = j$
$k = f$	$d = b$ and $g = a$	$g = b$ and $p = j$	$h = b$ and $k = j$	$k = e$ and $o = m$
$k = i$	$d = b$ and $g = b$	$g = b$ and $p = n$	$h = b$ and $o = f$	$k = e$ and $o = n$
$k = j$	$d = b$ and $g = d$	$g = d$ and $h = b$	$h = b$ and $o = i$	$k = e$ and $l = f$
$o = f$	$d = b$ and $g = e$	$g = d$ and $h = e$	$h = b$ and $o = j$	$k = e$ and $l = i$
$o = i$	$d = b$ and $g = f$	$g = d$ and $h = f$	$h = b$ and $o = m$	$k = e$ and $l = j$
$o = j$	$d = b$ and $h = e$	$g = d$ and $k = b$	$h = b$ and $o = n$	$k = e$ and $m = j$
$o = m$	$d = b$ and $h = f$	$g = d$ and $k = e$	$h = b$ and $l = f$	$k = e$ and $p = j$
$o = n$	$d = b$ and $k = e$	$g = d$ and $k = f$	$h = b$ and $l = i$	$k = e$ and $p = n$
$l = f$	$d = b$ and $k = f$	$g = d$ and $k = i$	$h = b$ and $l = j$	$k = f$ and $k = j$
$l = i$	$d = b$ and $k = i$	$g = d$ and $k = j$	$h = b$ and $m = j$	$k = f$ and $o = f$
$l = j$	$d = b$ and $k = j$	$g = d$ and $o = f$	$h = b$ and $p = j$	$k = f$ and $o = i$
$m = j$	$d = b$ and $o = f$	$g = d$ and $o = i$	$h = b$ and $p = n$	$k = f$ and $o = j$
$p = j$	$d = b$ and $o = i$	$g = d$ and $o = j$	$h = e$ and $k = b$	$k = f$ and $o = m$
$c = a$ and $d = b$	$d = b$ and $o = m$	$g = d$ and $o = n$	$h = e$ and $k = i$	$k = f$ and $l = i$
$c = a$ and $g = b$	$d = b$ and $o = n$	$g = d$ and $l = f$	$h = e$ and $k = j$	$k = f$ and $l = j$
$c = a$ and $g = d$	$d = b$ and $l = f$	$g = d$ and $l = i$	$h = e$ and $o = f$	$k = f$ and $m = j$
$c = a$ and $g = e$	$d = b$ and $l = i$	$g = d$ and $l = j$	$h = e$ and $o = i$	$k = f$ and $p = j$
$c = a$ and $h = b$	$d = b$ and $l = j$	$g = d$ and $m = j$	$h = e$ and $o = j$	$k = f$ and $p = n$
$c = a$ and $h = e$	$d = b$ and $m = j$	$g = d$ and $p = j$	$h = e$ and $o = m$	$k = i$ and $o = f$
$c = a$ and $h = f$	$d = b$ and $p = j$	$g = d$ and $p = n$	$h = e$ and $o = n$	$k = i$ and $o = i$
$c = a$ and $k = b$	$d = b$ and $p = n$	$g = e$ and $h = b$	$h = e$ and $l = f$	$k = i$ and $o = j$
$c = a$ and $k = e$	$g = a$ and $g = e$	$g = e$ and $h = f$	$h = e$ and $l = i$	$k = i$ and $o = m$
$c = a$ and $k = f$	$g = a$ and $g = f$	$g = e$ and $k = b$	$h = e$ and $l = j$	$k = i$ and $o = n$
$c = a$ and $k = i$	$g = a$ and $h = b$	$g = e$ and $k = e$	$h = e$ and $m = j$	$k = i$ and $l = f$
$c = a$ and $k = j$	$g = a$ and $h = e$	$g = e$ and $k = f$	$h = e$ and $p = j$	$k = i$ and $l = j$
$c = a$ and $o = f$	$g = a$ and $h = f$	$g = e$ and $k = i$	$h = e$ and $p = n$	$k = i$ and $m = j$
$c = a$ and $o = i$	$g = a$ and $k = b$	$g = e$ and $k = j$	$h = f$ and $k = b$	$k = i$ and $p = j$
$c = a$ and $o = j$	$g = a$ and $k = e$	$g = e$ and $o = f$	$h = f$ and $k = e$	$k = i$ and $p = n$
$c = a$ and $o = m$	$g = a$ and $k = f$	$g = e$ and $o = i$	$h = f$ and $k = i$	$k = j$ and $o = f$
$c = a$ and $o = n$	$g = a$ and $k = i$	$g = e$ and $o = j$	$h = f$ and $k = j$	$k = j$ and $o = i$
$c = a$ and $l = f$	$g = a$ and $k = j$	$g = e$ and $o = m$	$h = f$ and $o = f$	$k = j$ and $o = j$
$c = a$ and $l = i$	$g = a$ and $o = f$	$g = e$ and $o = n$	$h = f$ and $o = i$	$k = j$ and $o = m$
$c = a$ and $l = j$	$g = a$ and $o = i$	$g = e$ and $l = f$	$h = f$ and $o = j$	$k = j$ and $o = n$
$c = a$ and $m = j$	$g = a$ and $o = j$	$g = e$ and $l = i$	$h = f$ and $o = m$	$k = j$ and $l = f$
$c = a$ and $p = j$	$g = a$ and $o = m$	$g = e$ and $l = j$	$h = f$ and $o = n$	$k = j$ and $l = i$
$c = a$ and $p = n$	$g = a$ and $o = n$	$g = e$ and $m = j$	$h = f$ and $l = f$	$k = j$ and $p = j$
$c = a$ and $g = f$	$g = a$ and $l = f$	$g = e$ and $p = j$	$h = f$ and $l = i$	$k = j$ and $p = n$
$c = b$ and $g = a$	$g = a$ and $l = i$	$g = e$ and $p = n$	$h = f$ and $l = j$	$o = f$ and $o = j$
$c = b$ and $g = d$	$g = a$ and $l = j$	$g = f$ and $h = b$	$h = f$ and $m = j$	$o = f$ and $o = n$
$c = b$ and $g = e$	$g = a$ and $m = j$	$g = f$ and $h = e$	$h = f$ and $p = j$	$o = f$ and $l = i$
$c = b$ and $g = f$	$g = a$ and $p = j$	$g = f$ and $k = b$	$h = f$ and $p = n$	$o = f$ and $l = j$
$c = b$ and $h = b$	$g = a$ and $p = n$	$g = f$ and $k = e$	$k = b$ and $k = f$	$o = f$ and $m = j$
$c = b$ and $h = e$	$g = b$ and $g = d$	$g = f$ and $k = f$	$k = b$ and $k = i$	$o = f$ and $p = j$
$c = b$ and $h = f$	$g = b$ and $g = f$	$g = f$ and $k = i$	$k = b$ and $k = j$	$o = f$ and $p = n$
$c = b$ and $k = b$	$g = b$ and $h = e$	$g = f$ and $k = j$	$k = b$ and $o = f$	$o = i$ and $o = n$
$c = b$ and $k = e$	$g = b$ and $h = f$	$g = f$ and $o = f$	$k = b$ and $o = i$	$o = i$ and $l = f$
$c = b$ and $k = f$	$g = b$ and $k = b$	$g = f$ and $o = i$	$k = b$ and $o = j$	$o = i$ and $l = j$

$o = i$ and $m = j$	$c = a$, $g = b$ and $o = f$	$c = a$, $h = b$ and $m = j$	$c = a$, $k = f$ and $o = f$	$c = a$, $l = i$ and $m = j$
$o = i$ and $p = j$	$c = a$, $g = b$ and $o = i$	$c = a$, $h = b$ and $p = j$	$c = a$, $k = f$ and $o = i$	$c = a$, $l = i$ and $p = j$
$o = i$ and $p = n$	$c = a$, $g = b$ and $o = j$	$c = a$, $h = b$ and $p = n$	$c = a$, $k = f$ and $o = j$	$c = a$, $l = i$ and $p = n$
$o = j$ and $o = m$	$c = a$, $g = b$ and $o = m$	$c = a$, $h = e$ and $k = b$	$c = a$, $k = f$ and $o = m$	$c = a$, $l = j$ and $p = j$
$o = j$ and $l = f$	$c = a$, $g = b$ and $o = n$	$c = a$, $h = e$ and $k = f$	$c = a$, $k = f$ and $o = n$	$c = a$, $l = j$ and $p = n$
$o = j$ and $l = i$	$c = a$, $g = b$ and $l = f$	$c = a$, $h = e$ and $k = i$	$c = a$, $k = f$ and $l = j$	$c = a$, $m = j$ and $p = n$
$o = j$ and $m = j$	$c = a$, $g = b$ and $l = i$	$c = a$, $h = e$ and $k = j$	$c = a$, $k = f$ and $m = j$	$c = a$, $g = f$ and $h = b$
$o = j$ and $p = n$	$c = a$, $g = b$ and $l = j$	$c = a$, $h = e$ and $o = f$	$c = a$, $k = f$ and $p = j$	$c = a$, $g = f$ and $h = e$
$o = m$ and $l = f$	$c = a$, $g = b$ and $m = j$	$c = a$, $h = e$ and $o = i$	$c = a$, $k = f$ and $p = n$	$c = a$, $g = f$ and $k = b$
$o = m$ and $l = i$	$c = a$, $g = b$ and $p = j$	$c = a$, $h = e$ and $o = j$	$c = a$, $k = i$ and $o = f$	$c = a$, $g = f$ and $k = e$
$o = m$ and $l = j$	$c = a$, $g = b$ and $p = n$	$c = a$, $h = e$ and $o = m$	$c = a$, $k = i$ and $o = i$	$c = a$, $g = f$ and $k = f$
$o = m$ and $m = j$	$c = a$, $g = d$ and $h = b$	$c = a$, $h = e$ and $o = n$	$c = a$, $k = i$ and $o = j$	$c = a$, $g = f$ and $k = i$
$o = m$ and $p = j$	$c = a$, $g = d$ and $h = e$	$c = a$, $h = e$ and $l = f$	$c = a$, $k = i$ and $o = m$	$c = a$, $g = f$ and $k = j$
$o = m$ and $p = n$	$c = a$, $g = d$ and $h = f$	$c = a$, $h = e$ and $l = i$	$c = a$, $k = i$ and $o = n$	$c = a$, $g = f$ and $o = f$
$o = n$ and $l = f$	$c = a$, $g = d$ and $k = b$	$c = a$, $h = e$ and $l = j$	$c = a$, $k = i$ and $l = f$	$c = a$, $g = f$ and $o = i$
$o = n$ and $l = i$	$c = a$, $g = d$ and $k = e$	$c = a$, $h = e$ and $m = j$	$c = a$, $k = i$ and $l = j$	$c = a$, $g = f$ and $o = j$
$o = n$ and $l = j$	$c = a$, $g = d$ and $k = f$	$c = a$, $h = e$ and $p = j$	$c = a$, $k = i$ and $m = j$	$c = a$, $g = f$ and $o = m$
$o = n$ and $m = j$	$c = a$, $g = d$ and $k = i$	$c = a$, $h = e$ and $p = n$	$c = a$, $k = i$ and $p = j$	$c = a$, $g = f$ and $o = n$
$o = n$ and $p = j$	$c = a$, $g = d$ and $k = j$	$c = a$, $h = f$ and $k = b$	$c = a$, $k = i$ and $p = n$	$c = a$, $g = f$ and $l = f$
$o = n$ and $p = n$	$c = a$, $g = d$ and $o = f$	$c = a$, $h = f$ and $k = e$	$c = a$, $k = j$ and $o = f$	$c = a$, $g = f$ and $l = i$
$l = f$ and $l = j$	$c = a$, $g = d$ and $o = i$	$c = a$, $h = f$ and $k = i$	$c = a$, $k = j$ and $o = i$	$c = a$, $g = f$ and $l = j$
$l = f$ and $m = j$	$c = a$, $g = d$ and $o = j$	$c = a$, $h = f$ and $k = j$	$c = a$, $k = j$ and $o = j$	$c = a$, $g = f$ and $m = j$
$l = f$ and $p = j$	$c = a$, $g = d$ and $o = m$	$c = a$, $h = f$ and $o = f$	$c = a$, $k = j$ and $o = m$	$c = a$, $g = f$ and $p = j$
$l = f$ and $p = n$	$c = a$, $g = d$ and $o = n$	$c = a$, $h = f$ and $o = i$	$c = a$, $k = j$ and $o = n$	$c = a$, $g = f$ and $p = n$
$l = f$ and $m = j$	$c = a$, $g = d$ and $l = f$	$c = a$, $h = f$ and $o = j$	$c = a$, $k = j$ and $l = f$	$c = b$, $g = a$ and $g = e$
$l = f$ and $p = j$	$c = a$, $g = d$ and $l = i$	$c = a$, $h = f$ and $o = m$	$c = a$, $k = j$ and $l = i$	$c = b$, $g = a$ and $g = f$
$l = f$ and $p = n$	$c = a$, $g = d$ and $l = j$	$c = a$, $h = f$ and $o = n$	$c = a$, $k = j$ and $p = j$	$c = b$, $g = a$ and $h = b$
$l = f$ and $p = j$	$c = a$, $g = d$ and $m = j$	$c = a$, $h = f$ and $l = f$	$c = a$, $k = j$ and $p = n$	$c = b$, $g = a$ and $h = e$
$l = f$ and $p = n$	$c = a$, $g = d$ and $p = j$	$c = a$, $h = f$ and $l = i$	$c = a$, $o = f$ and $o = j$	$c = b$, $g = a$ and $h = f$
$m = j$ and $p = n$	$c = a$, $g = d$ and $p = n$	$c = a$, $h = f$ and $l = j$	$c = a$, $o = f$ and $o = n$	$c = b$, $g = a$ and $k = b$
$p = j$ and $p = n$	$c = a$, $g = e$ and $h = b$	$c = a$, $h = f$ and $m = j$	$c = a$, $o = f$ and $l = i$	$c = b$, $g = a$ and $k = e$
$c = a$, $d = b$ and $g = b$	$c = a$, $g = e$ and $h = f$	$c = a$, $h = f$ and $p = j$	$c = a$, $o = f$ and $l = j$	$c = b$, $g = a$ and $k = f$
$c = a$, $d = b$ and $g = d$	$c = a$, $g = e$ and $k = b$	$c = a$, $h = f$ and $p = n$	$c = a$, $o = f$ and $m = j$	$c = b$, $g = a$ and $k = i$
$c = a$, $d = b$ and $g = e$	$c = a$, $g = e$ and $k = e$	$c = a$, $k = b$ and $k = f$	$c = a$, $o = f$ and $p = j$	$c = b$, $g = a$ and $k = j$
$c = a$, $d = b$ and $g = f$	$c = a$, $g = e$ and $k = f$	$c = a$, $k = b$ and $k = i$	$c = a$, $o = f$ and $p = n$	$c = b$, $g = a$ and $o = f$
$c = a$, $d = b$ and $h = e$	$c = a$, $g = e$ and $k = i$	$c = a$, $k = b$ and $k = j$	$c = a$, $o = i$ and $o = n$	$c = b$, $g = a$ and $o = i$
$c = a$, $d = b$ and $h = f$	$c = a$, $g = e$ and $k = j$	$c = a$, $k = b$ and $o = f$	$c = a$, $o = i$ and $l = f$	$c = b$, $g = a$ and $o = j$
$c = a$, $d = b$ and $k = e$	$c = a$, $g = e$ and $o = f$	$c = a$, $k = b$ and $o = i$	$c = a$, $o = i$ and $l = j$	$c = b$, $g = a$ and $o = m$
$c = a$, $d = b$ and $k = f$	$c = a$, $g = e$ and $o = i$	$c = a$, $k = b$ and $o = j$	$c = a$, $o = i$ and $m = j$	$c = b$, $g = a$ and $o = n$
$c = a$, $d = b$ and $k = i$	$c = a$, $g = e$ and $o = j$	$c = a$, $k = b$ and $o = m$	$c = a$, $o = i$ and $p = j$	$c = b$, $g = a$ and $l = f$
$c = a$, $d = b$ and $k = j$	$c = a$, $g = e$ and $o = m$	$c = a$, $k = b$ and $o = n$	$c = a$, $o = i$ and $p = n$	$c = b$, $g = a$ and $l = i$
$c = a$, $d = b$ and $o = f$	$c = a$, $g = e$ and $o = n$	$c = a$, $k = b$ and $l = f$	$c = a$, $o = j$ and $o = m$	$c = b$, $g = a$ and $l = j$
$c = a$, $d = b$ and $o = i$	$c = a$, $g = e$ and $l = f$	$c = a$, $k = b$ and $l = i$	$c = a$, $o = j$ and $l = f$	$c = b$, $g = a$ and $m = j$
$c = a$, $d = b$ and $o = j$	$c = a$, $g = e$ and $l = i$	$c = a$, $k = b$ and $l = j$	$c = a$, $o = j$ and $l = j$	$c = b$, $g = a$ and $p = j$
$c = a$, $d = b$ and $o = m$	$c = a$, $g = e$ and $l = j$	$c = a$, $k = b$ and $m = j$	$c = a$, $o = j$ and $m = j$	$c = b$, $g = a$ and $p = n$
$c = a$, $d = b$ and $o = n$	$c = a$, $g = e$ and $m = j$	$c = a$, $k = b$ and $p = j$	$c = a$, $o = j$ and $p = n$	$c = b$, $g = d$ and $h = b$
$c = a$, $d = b$ and $l = f$	$c = a$, $g = e$ and $p = j$	$c = a$, $k = b$ and $p = n$	$c = a$, $o = m$ and $l = f$	$c = b$, $g = d$ and $h = e$
$c = a$, $d = b$ and $l = i$	$c = a$, $g = e$ and $p = n$	$c = a$, $k = e$ and $k = i$	$c = a$, $o = m$ and $l = i$	$c = b$, $g = d$ and $h = f$
$c = a$, $d = b$ and $l = j$	$c = a$, $h = b$ and $h = f$	$c = a$, $k = e$ and $k = j$	$c = a$, $o = m$ and $l = j$	$c = b$, $g = d$ and $k = b$
$c = a$, $d = b$ and $m = j$	$c = a$, $h = b$ and $k = e$	$c = a$, $k = e$ and $o = f$	$c = a$, $o = m$ and $m = j$	$c = b$, $g = d$ and $k = e$
$c = a$, $d = b$ and $p = j$	$c = a$, $h = b$ and $k = f$	$c = a$, $k = e$ and $o = i$	$c = a$, $o = m$ and $p = j$	$c = b$, $g = d$ and $k = f$
$c = a$, $d = b$ and $p = n$	$c = a$, $h = b$ and $k = i$	$c = a$, $k = e$ and $o = j$	$c = a$, $o = m$ and $p = n$	$c = b$, $g = d$ and $k = i$
$c = a$, $d = b$ and $g = d$	$c = a$, $h = b$ and $k = j$	$c = a$, $k = e$ and $o = m$	$c = a$, $o = n$ and $l = f$	$c = b$, $g = d$ and $k = j$
$c = a$, $d = b$ and $g = f$	$c = a$, $h = b$ and $o = f$	$c = a$, $k = e$ and $o = n$	$c = a$, $o = n$ and $l = i$	$c = b$, $g = d$ and $o = f$
$c = a$, $d = b$ and $h = e$	$c = a$, $h = b$ and $o = i$	$c = a$, $k = e$ and $l = f$	$c = a$, $o = n$ and $l = j$	$c = b$, $g = d$ and $o = i$
$c = a$, $d = b$ and $h = f$	$c = a$, $h = b$ and $o = j$	$c = a$, $k = e$ and $l = i$	$c = a$, $o = n$ and $m = j$	$c = b$, $g = d$ and $o = j$
$c = a$, $d = b$ and $k = b$	$c = a$, $h = b$ and $o = m$	$c = a$, $k = e$ and $l = j$	$c = a$, $o = n$ and $p = j$	$c = b$, $g = d$ and $o = m$
$c = a$, $d = b$ and $k = e$	$c = a$, $h = b$ and $o = n$	$c = a$, $k = e$ and $m = j$	$c = a$, $l = f$ and $l = j$	$c = b$, $g = d$ and $o = n$
$c = a$, $d = b$ and $k = f$	$c = a$, $h = b$ and $l = f$	$c = a$, $k = e$ and $p = j$	$c = a$, $l = f$ and $m = j$	$c = b$, $g = d$ and $l = f$
$c = a$, $d = b$ and $k = i$	$c = a$, $h = b$ and $l = i$	$c = a$, $k = e$ and $p = n$	$c = a$, $l = f$ and $p = j$	$c = b$, $g = d$ and $l = i$
$c = a$, $d = b$ and $k = j$	$c = a$, $h = b$ and $l = j$	$c = a$, $k = f$ and $k = j$	$c = a$, $l = f$ and $p = n$	$c = b$, $g = d$ and $l = j$

$c = b$, $g = d$ and $m = j$	$c = b$, $h = e$ and $o = j$	$c = a$, $k = f$ and $p = n$	$d = b$, $g = a$ and $g = e$	$d = b$, $g = f$ and $o = m$
$c = b$, $g = d$ and $p = j$	$c = b$, $h = e$ and $o = m$	$c = b$, $k = i$ and $o = f$	$d = b$, $g = a$ and $g = f$	$d = b$, $g = f$ and $o = n$
$c = b$, $g = d$ and $p = n$	$c = b$, $h = e$ and $o = n$	$c = b$, $k = i$ and $o = i$	$d = b$, $g = a$ and $h = e$	$d = b$, $g = f$ and $l = f$
$c = b$, $g = e$ and $h = b$	$c = b$, $h = e$ and $l = f$	$c = b$, $k = i$ and $o = j$	$d = b$, $g = a$ and $h = f$	$d = b$, $g = f$ and $l = i$
$c = b$, $g = e$ and $h = f$	$c = b$, $h = e$ and $l = i$	$c = b$, $k = i$ and $o = m$	$d = b$, $g = a$ and $k = e$	$d = b$, $g = f$ and $l = j$
$c = b$, $g = e$ and $k = b$	$c = b$, $h = e$ and $l = j$	$c = b$, $k = i$ and $o = n$	$d = b$, $g = a$ and $k = f$	$d = b$, $g = f$ and $m = j$
$c = b$, $g = e$ and $k = e$	$c = b$, $h = e$ and $m = j$	$c = b$, $k = i$ and $l = f$	$d = b$, $g = a$ and $k = i$	$d = b$, $g = f$ and $p = j$
$c = b$, $g = e$ and $k = f$	$c = b$, $h = e$ and $p = j$	$c = b$, $k = i$ and $l = i$	$d = b$, $g = a$ and $k = j$	$d = b$, $g = f$ and $p = n$
$c = b$, $g = e$ and $k = i$	$c = b$, $h = e$ and $p = n$	$c = b$, $k = i$ and $l = j$	$d = b$, $g = a$ and $o = f$	$d = b$, $h = e$ and $k = f$
$c = b$, $g = e$ and $k = j$	$c = b$, $h = f$ and $k = b$	$c = b$, $k = i$ and $m = j$	$d = b$, $g = a$ and $o = i$	$d = b$, $h = e$ and $k = i$
$c = b$, $g = e$ and $o = i$	$c = b$, $h = f$ and $k = i$	$c = b$, $k = i$ and $p = n$	$d = b$, $g = a$ and $o = m$	$d = b$, $h = e$ and $o = f$
$c = b$, $g = e$ and $o = j$	$c = b$, $h = f$ and $k = j$	$c = b$, $k = j$ and $o = f$	$d = b$, $g = a$ and $o = n$	$d = b$, $h = e$ and $o = i$
$c = b$, $g = e$ and $o = m$	$c = b$, $h = f$ and $o = f$	$c = b$, $k = j$ and $o = i$	$d = b$, $g = a$ and $l = f$	$d = b$, $h = e$ and $o = j$
$c = b$, $g = e$ and $o = n$	$c = b$, $h = f$ and $o = i$	$c = b$, $k = j$ and $o = j$	$d = b$, $g = a$ and $l = i$	$d = b$, $h = e$ and $o = m$
$c = b$, $g = e$ and $l = f$	$c = b$, $h = f$ and $o = j$	$c = b$, $k = j$ and $o = m$	$d = b$, $g = a$ and $l = j$	$d = b$, $h = e$ and $o = n$
$c = b$, $g = e$ and $l = i$	$c = b$, $h = f$ and $o = m$	$c = b$, $k = j$ and $o = n$	$d = b$, $g = a$ and $m = j$	$d = b$, $h = e$ and $l = f$
$c = b$, $g = e$ and $l = j$	$c = b$, $h = f$ and $o = n$	$c = b$, $k = j$ and $l = f$	$d = b$, $g = a$ and $p = j$	$d = b$, $h = e$ and $l = i$
$c = b$, $g = e$ and $m = j$	$c = b$, $h = f$ and $l = f$	$c = b$, $k = j$ and $l = i$	$d = b$, $g = a$ and $p = n$	$d = b$, $h = e$ and $l = j$
$c = b$, $g = e$ and $p = j$	$c = b$, $h = f$ and $l = i$	$c = b$, $k = j$ and $p = j$	$d = b$, $g = b$ and $g = d$	$d = b$, $h = e$ and $m = j$
$c = b$, $g = e$ and $p = n$	$c = b$, $h = f$ and $l = j$	$c = b$, $k = j$ and $p = n$	$d = b$, $g = d$ and $h = e$	$d = b$, $h = e$ and $p = j$
$c = b$, $g = f$ and $h = b$	$c = b$, $h = f$ and $m = j$	$c = b$, $o = f$ and $o = j$	$d = b$, $g = d$ and $h = f$	$d = b$, $h = e$ and $p = n$
$c = b$, $g = f$ and $h = e$	$c = b$, $h = f$ and $p = j$	$c = b$, $o = f$ and $o = n$	$d = b$, $g = d$ and $k = e$	$d = b$, $h = f$ and $k = e$
$c = b$, $g = f$ and $k = b$	$c = b$, $h = f$ and $p = n$	$c = b$, $o = f$ and $l = i$	$d = b$, $g = d$ and $k = f$	$d = b$, $h = f$ and $k = i$
$c = b$, $g = f$ and $k = e$	$c = b$, $k = b$ and $k = f$	$c = b$, $o = f$ and $l = j$	$d = b$, $g = d$ and $k = i$	$d = b$, $h = f$ and $k = j$
$c = b$, $g = f$ and $k = f$	$c = b$, $k = b$ and $k = i$	$c = b$, $o = f$ and $m = j$	$d = b$, $g = d$ and $k = j$	$d = b$, $h = f$ and $o = f$
$c = b$, $g = f$ and $k = i$	$c = b$, $k = b$ and $k = j$	$c = b$, $o = f$ and $p = j$	$d = b$, $g = d$ and $o = f$	$d = b$, $h = f$ and $o = i$
$c = b$, $g = f$ and $k = j$	$c = b$, $k = b$ and $o = f$	$c = b$, $o = f$ and $p = n$	$d = b$, $g = d$ and $o = i$	$d = b$, $h = f$ and $o = j$
$c = b$, $g = f$ and $o = f$	$c = b$, $k = b$ and $o = i$	$c = b$, $o = i$ and $o = n$	$d = b$, $g = d$ and $o = j$	$d = b$, $h = f$ and $o = m$
$c = b$, $g = f$ and $o = i$	$c = b$, $k = b$ and $o = j$	$c = b$, $o = i$ and $l = f$	$d = b$, $g = d$ and $o = m$	$d = b$, $h = f$ and $o = n$
$c = b$, $g = f$ and $o = j$	$c = b$, $k = b$ and $o = m$	$c = b$, $o = i$ and $l = j$	$d = b$, $g = d$ and $o = n$	$d = b$, $h = f$ and $l = f$
$c = b$, $g = f$ and $o = m$	$c = b$, $k = b$ and $o = n$	$c = b$, $o = i$ and $m = j$	$d = b$, $g = d$ and $l = f$	$d = b$, $h = f$ and $l = i$
$c = b$, $g = f$ and $o = n$	$c = b$, $k = b$ and $l = f$	$c = b$, $o = i$ and $p = j$	$d = b$, $g = d$ and $l = i$	$d = b$, $h = f$ and $l = j$
$c = b$, $g = f$ and $l = f$	$c = b$, $k = b$ and $l = i$	$c = b$, $o = i$ and $p = n$	$d = b$, $g = d$ and $l = j$	$d = b$, $h = f$ and $m = j$
$c = b$, $g = f$ and $l = i$	$c = b$, $k = b$ and $l = j$	$c = b$, $o = j$ and $o = m$	$d = b$, $g = d$ and $m = j$	$d = b$, $h = f$ and $p = j$
$c = b$, $g = f$ and $l = j$	$c = b$, $k = b$ and $m = j$	$c = b$, $o = j$ and $l = f$	$d = b$, $g = d$ and $p = j$	$d = b$, $h = f$ and $p = n$
$c = b$, $g = f$ and $m = j$	$c = b$, $k = b$ and $p = j$	$c = b$, $o = j$ and $l = i$	$d = b$, $g = d$ and $p = n$	$d = b$, $k = e$ and $k = i$
$c = b$, $g = f$ and $p = j$	$c = b$, $k = b$ and $p = n$	$c = b$, $o = j$ and $m = j$	$d = b$, $g = e$ and $h = f$	$d = b$, $k = e$ and $k = j$
$c = b$, $g = f$ and $p = n$	$c = b$, $k = e$ and $k = i$	$c = b$, $o = j$ and $p = j$	$d = b$, $g = e$ and $k = e$	$d = b$, $k = e$ and $o = f$
$c = b$, $h = b$ and $h = f$	$c = b$, $k = e$ and $k = j$	$c = b$, $o = j$ and $p = n$	$d = b$, $g = e$ and $k = f$	$d = b$, $k = e$ and $o = i$
$c = b$, $h = b$ and $k = e$	$c = b$, $k = e$ and $o = f$	$c = b$, $o = m$ and $l = f$	$d = b$, $g = e$ and $k = i$	$d = b$, $k = e$ and $o = j$
$c = b$, $h = b$ and $k = f$	$c = b$, $k = e$ and $o = i$	$c = b$, $o = m$ and $l = i$	$d = b$, $g = e$ and $k = j$	$d = b$, $k = e$ and $o = m$
$c = b$, $h = b$ and $k = i$	$c = b$, $k = e$ and $o = j$	$c = b$, $o = m$ and $l = j$	$d = b$, $g = e$ and $o = f$	$d = b$, $k = e$ and $o = n$
$c = b$, $h = b$ and $k = j$	$c = b$, $k = e$ and $o = m$	$c = b$, $o = m$ and $m = j$	$d = b$, $g = e$ and $o = i$	$d = b$, $k = e$ and $l = f$
$c = b$, $h = b$ and $o = f$	$c = b$, $k = e$ and $o = n$	$c = b$, $o = m$ and $p = j$	$d = b$, $g = e$ and $o = j$	$d = b$, $k = e$ and $l = i$
$c = b$, $h = b$ and $o = i$	$c = b$, $k = e$ and $l = f$	$c = b$, $o = m$ and $p = n$	$d = b$, $g = e$ and $o = m$	$d = b$, $k = e$ and $l = j$
$c = b$, $h = b$ and $o = j$	$c = b$, $k = e$ and $l = i$	$c = b$, $o = n$ and $l = f$	$d = b$, $g = e$ and $o = n$	$d = b$, $k = e$ and $m = j$
$c = b$, $h = b$ and $o = m$	$c = b$, $k = e$ and $l = j$	$c = b$, $o = n$ and $l = i$	$d = b$, $g = e$ and $l = f$	$d = b$, $k = e$ and $p = j$
$c = b$, $h = b$ and $o = n$	$c = b$, $k = e$ and $m = j$	$c = b$, $o = n$ and $l = j$	$d = b$, $g = e$ and $l = i$	$d = b$, $k = e$ and $p = n$
$c = b$, $h = b$ and $l = f$	$c = b$, $k = e$ and $p = j$	$c = b$, $o = n$ and $m = j$	$d = b$, $g = e$ and $l = j$	$d = b$, $k = f$ and $k = j$
$c = b$, $h = b$ and $l = i$	$c = b$, $k = e$ and $p = n$	$c = b$, $o = n$ and $p = j$	$d = b$, $g = e$ and $m = j$	$d = b$, $k = f$ and $o = f$
$c = b$, $h = b$ and $l = j$	$c = b$, $k = f$ and $k = j$	$c = b$, $l = f$ and $l = j$	$d = b$, $g = e$ and $p = j$	$d = b$, $k = f$ and $o = i$
$c = b$, $h = b$ and $m = j$	$c = b$, $k = f$ and $o = f$	$c = b$, $l = f$ and $m = j$	$d = b$, $g = e$ and $p = n$	$d = b$, $k = f$ and $o = j$
$c = b$, $h = b$ and $p = j$	$c = b$, $k = f$ and $o = i$	$c = b$, $l = f$ and $p = j$	$d = b$, $g = f$ and $h = e$	$d = b$, $k = f$ and $o = m$
$c = b$, $h = b$ and $p = n$	$c = b$, $k = f$ and $o = j$	$c = b$, $l = f$ and $p = n$	$d = b$, $g = f$ and $k = e$	$d = b$, $k = f$ and $o = n$
$c = b$, $h = e$ and $k = b$	$c = b$, $k = f$ and $o = m$	$c = b$, $l = i$ and $m = j$	$d = b$, $g = f$ and $k = f$	$d = b$, $k = f$ and $l = f$
$c = b$, $h = e$ and $k = f$	$c = b$, $k = f$ and $o = n$	$c = b$, $l = i$ and $p = j$	$d = b$, $g = f$ and $k = i$	$d = b$, $k = f$ and $l = i$
$c = b$, $h = e$ and $k = i$	$c = b$, $k = f$ and $l = i$	$c = b$, $l = i$ and $p = n$	$d = b$, $g = f$ and $k = j$	$d = b$, $k = f$ and $l = j$
$c = b$, $h = e$ and $k = j$	$c = b$, $k = f$ and $l = j$	$c = b$, $l = j$ and $p = j$	$d = b$, $g = f$ and $o = f$	$d = b$, $k = f$ and $m = j$
$c = b$, $h = e$ and $o = f$	$c = b$, $k = f$ and $m = j$	$c = b$, $l = j$ and $p = n$	$d = b$, $g = f$ and $o = i$	$d = b$, $k = f$ and $p = j$
$c = b$, $h = e$ and $o = i$	$c = b$, $k = f$ and $p = j$	$c = b$, $m = j$ and $p = n$	$d = b$, $g = f$ and $o = j$	$d = b$, $k = f$ and $p = n$

$d = b$, $k = i$ and $o = f$	$g = a$, $g = e$ and $k = e$	$g = a$, $h = e$ and $m = j$	$g = a$, $k = i$ and $o = n$
$d = b$, $k = i$ and $o = i$	$g = a$, $g = e$ and $k = f$	$g = a$, $h = e$ and $p = j$	$g = a$, $k = i$ and $l = f$
$d = b$, $k = i$ and $o = j$	$g = a$, $g = e$ and $k = i$	$g = a$, $h = e$ and $p = n$	$g = a$, $k = i$ and $l = j$
$d = b$, $k = i$ and $o = m$	$g = a$, $g = e$ and $k = j$	$g = a$, $h = f$ and $k = b$	$g = a$, $k = i$ and $m = j$
$d = b$, $k = i$ and $o = n$	$g = a$, $g = e$ and $o = f$	$g = a$, $h = f$ and $k = e$	$g = a$, $k = i$ and $p = j$
$d = b$, $k = i$ and $l = f$	$g = a$, $g = e$ and $o = i$	$g = a$, $h = f$ and $k = f$	$g = a$, $k = i$ and $p = n$
$d = b$, $k = i$ and $l = j$	$g = a$, $g = e$ and $o = j$	$g = a$, $h = f$ and $k = i$	$g = a$, $k = j$ and $o = f$
$d = b$, $k = i$ and $m = j$	$g = a$, $g = e$ and $o = m$	$g = a$, $h = f$ and $k = j$	$g = a$, $k = j$ and $o = i$
$d = b$, $k = i$ and $p = j$	$g = a$, $g = e$ and $o = n$	$g = a$, $h = f$ and $o = f$	$g = a$, $k = j$ and $o = j$
$d = b$, $k = i$ and $p = n$	$g = a$, $g = e$ and $l = f$	$g = a$, $h = f$ and $o = i$	$g = a$, $k = j$ and $o = m$
$d = b$, $k = j$ and $o = f$	$g = a$, $g = e$ and $l = i$	$g = a$, $h = f$ and $o = j$	$g = a$, $k = j$ and $o = n$
$d = b$, $k = j$ and $o = i$	$g = a$, $g = e$ and $l = j$	$g = a$, $h = f$ and $o = m$	$g = a$, $k = j$ and $l = f$
$d = b$, $k = j$ and $o = j$	$g = a$, $g = e$ and $m = j$	$g = a$, $h = f$ and $o = n$	$g = a$, $k = j$ and $l = i$
$d = b$, $k = j$ and $o = m$	$g = a$, $g = e$ and $p = j$	$g = a$, $h = f$ and $l = f$	$g = a$, $k = j$ and $p = j$
$d = b$, $k = j$ and $o = n$	$g = a$, $g = e$ and $p = n$	$g = a$, $h = f$ and $l = i$	$g = a$, $k = j$ and $p = n$
$d = b$, $k = j$ and $l = f$	$g = a$, $g = f$ and $h = b$	$g = a$, $h = f$ and $l = j$	$g = a$, $o = f$ and $o = j$
$d = b$, $k = j$ and $l = i$	$g = a$, $g = f$ and $h = e$	$g = a$, $h = f$ and $m = j$	$g = a$, $o = f$ and $o = n$
$d = b$, $k = j$ and $p = j$	$g = a$, $g = f$ and $k = b$	$g = a$, $h = f$ and $p = j$	$g = a$, $o = f$ and $l = i$
$d = b$, $k = j$ and $p = n$	$g = a$, $g = f$ and $k = e$	$g = a$, $h = f$ and $p = n$	$g = a$, $o = f$ and $l = j$
$d = b$, $o = f$ and $o = j$	$g = a$, $g = f$ and $k = f$	$g = a$, $k = b$ and $k = f$	$g = a$, $o = f$ and $m = j$
$d = b$, $o = f$ and $o = n$	$g = a$, $g = f$ and $k = i$	$g = a$, $k = b$ and $k = i$	$g = a$, $o = f$ and $p = j$
$d = b$, $o = f$ and $l = i$	$g = a$, $g = f$ and $k = j$	$g = a$, $k = b$ and $k = j$	$g = a$, $o = f$ and $p = n$
$d = b$, $o = f$ and $l = j$	$g = a$, $g = f$ and $o = f$	$g = a$, $k = b$ and $o = f$	$g = a$, $o = i$ and $o = n$
$d = b$, $o = f$ and $m = j$	$g = a$, $g = f$ and $o = i$	$g = a$, $k = b$ and $o = i$	$g = a$, $o = i$ and $l = f$
$d = b$, $o = f$ and $p = j$	$g = a$, $g = f$ and $o = j$	$g = a$, $k = b$ and $o = j$	$g = a$, $o = i$ and $l = j$
$d = b$, $o = f$ and $p = n$	$g = a$, $g = f$ and $o = m$	$g = a$, $k = b$ and $o = m$	$g = a$, $o = i$ and $m = j$
$d = b$, $o = i$ and $o = n$	$g = a$, $g = f$ and $o = n$	$g = a$, $k = b$ and $o = n$	$g = a$, $o = i$ and $p = j$
$d = b$, $o = i$ and $l = f$	$g = a$, $g = f$ and $l = f$	$g = a$, $k = b$ and $l = f$	$g = a$, $o = i$ and $p = n$
$d = b$, $o = i$ and $l = j$	$g = a$, $g = f$ and $l = i$	$g = a$, $k = b$ and $l = i$	$g = a$, $o = j$ and $o = m$
$d = b$, $o = i$ and $m = j$	$g = a$, $g = f$ and $l = j$	$g = a$, $k = b$ and $l = j$	$g = a$, $o = j$ and $l = f$
$d = b$, $o = i$ and $p = j$	$g = a$, $g = f$ and $m = j$	$g = a$, $k = b$ and $m = j$	$g = a$, $o = j$ and $l = i$
$d = b$, $o = i$ and $p = n$	$g = a$, $g = f$ and $p = j$	$g = a$, $k = b$ and $p = j$	$g = a$, $o = j$ and $m = j$
$d = b$, $o = j$ and $o = m$	$g = a$, $g = f$ and $p = n$	$g = a$, $k = b$ and $p = n$	$g = a$, $o = j$ and $p = n$
$d = b$, $o = j$ and $l = f$	$g = a$, $h = b$ and $h = f$	$g = a$, $k = e$ and $k = i$	$g = a$, $o = m$ and $l = f$
$d = b$, $o = j$ and $l = i$	$g = a$, $h = b$ and $k = e$	$g = a$, $k = e$ and $k = j$	$g = a$, $o = m$ and $l = i$
$d = b$, $o = j$ and $m = j$	$g = a$, $h = b$ and $k = f$	$g = a$, $k = e$ and $o = f$	$g = a$, $o = m$ and $l = j$
$d = b$, $o = j$ and $p = n$	$g = a$, $h = b$ and $k = i$	$g = a$, $k = e$ and $o = i$	$g = a$, $o = m$ and $m = j$
$d = b$, $o = m$ and $l = f$	$g = a$, $h = b$ and $k = j$	$g = a$, $k = e$ and $o = j$	$g = a$, $o = m$ and $p = j$

$d = b$, $o = m$ and $l = i$	$g = a$, $h = b$ and $o = f$	$g = a$, $k = e$ and $o = m$	$g = a$, $o = m$ and $p = n$
$d = b$, $o = m$ and $l = j$	$g = a$, $h = b$ and $o = i$	$g = a$, $k = e$ and $o = n$	$g = a$, $o = n$ and $l = f$
$d = b$, $o = m$ and $m = j$	$g = a$, $h = b$ and $o = j$	$g = a$, $k = e$ and $l = f$	$g = a$, $o = n$ and $l = i$
$d = b$, $o = m$ and $p = j$	$g = a$, $h = b$ and $o = m$	$g = a$, $k = e$ and $l = i$	$g = a$, $o = n$ and $l = j$
$d = b$, $o = m$ and $p = n$	$g = a$, $h = b$ and $o = n$	$g = a$, $k = e$ and $l = j$	$g = a$, $o = n$ and $m = j$
$d = b$, $o = n$ and $l = f$	$g = a$, $h = b$ and $l = f$	$g = a$, $k = e$ and $l = m$	$g = a$, $o = n$ and $p = j$
$d = b$, $o = n$ and $l = i$	$g = a$, $h = b$ and $l = i$	$g = a$, $k = e$ and $p = j$	$g = a$, $l = f$ and $l = j$
$d = b$, $o = n$ and $l = j$	$g = a$, $h = b$ and $l = j$	$g = a$, $k = e$ and $p = n$	$g = a$, $l = f$ and $m = j$
$d = b$, $o = n$ and $m = j$	$g = a$, $h = b$ and $m = j$	$g = a$, $k = f$ and $k = j$	$g = a$, $l = f$ and $p = j$
$d = b$, $o = n$ and $p = j$	$g = a$, $h = b$ and $p = j$	$g = a$, $k = f$ and $o = f$	$g = a$, $l = f$ and $p = n$
$d = b$, $l = f$ and $l = j$	$g = a$, $h = b$ and $p = n$	$g = a$, $k = f$ and $o = i$	$g = a$, $l = i$ and $m = j$
$d = b$, $l = f$ and $m = j$	$g = a$, $h = e$ and $k = b$	$g = a$, $k = f$ and $o = j$	$g = a$, $l = i$ and $p = j$
$d = b$, $l = f$ and $p = j$	$g = a$, $h = e$ and $k = f$	$g = a$, $k = f$ and $o = m$	$g = a$, $l = i$ and $p = n$
$d = b$, $l = f$ and $p = n$	$g = a$, $h = e$ and $k = i$	$g = a$, $k = f$ and $o = n$	$g = a$, $l = j$ and $p = j$
$d = b$, $l = i$ and $m = j$	$g = a$, $h = e$ and $k = j$	$g = a$, $k = f$ and $l = i$	$g = a$, $l = j$ and $p = n$
$d = b$, $l = i$ and $p = j$	$g = a$, $h = e$ and $o = f$	$g = a$, $k = f$ and $l = j$	$g = a$, $m = j$ and $p = n$
$d = b$, $l = i$ and $p = n$	$g = a$, $h = e$ and $o = i$	$g = a$, $k = f$ and $m = j$	$g = b$, $g = d$ and $d = b$
$d = b$, $l = i$ and $p = j$	$g = a$, $h = e$ and $o = j$	$g = a$, $k = f$ and $p = j$	$g = b$, $g = f$ and $b = f$
$d = b$, $l = i$ and $p = n$	$g = a$, $h = e$ and $o = m$	$g = a$, $k = f$ and $p = n$	$g = b$, $h = e$ and $k = b$
$d = b$, $m = j$ and $p = n$	$g = a$, $h = e$ and $o = n$	$g = a$, $k = i$ and $o = f$	$g = b$, $h = e$ and $k = f$
$g = a$, $g = e$ and $h = b$	$g = a$, $h = e$ and $l = f$	$g = a$, $k = i$ and $o = i$	$g = b$, $h = e$ and $k = i$
$g = a$, $g = e$ and $h = f$	$g = a$, $h = e$ and $l = i$	$g = a$, $k = i$ and $o = j$	$g = b$, $h = e$ and $k = j$
$g = b$, $h = e$ and $o = i$	$g = b$, $h = e$ and $o = j$	$g = b$, $h = e$ and $o = m$	$g = b$, $h = e$ and $o = n$
$g = b$, $h = e$ and $l = f$	$g = b$, $h = e$ and $l = i$	$g = b$, $h = e$ and $l = j$	$g = b$, $h = e$ and $m = j$
$g = b$, $h = e$ and $p = j$	$g = b$, $h = e$ and $p = n$	$g = b$, $h = f$ and $k = b$	$g = b$, $h = f$ and $k = e$
$g = b$, $h = f$ and $k = i$	$g = b$, $h = f$ and $k = j$	$g = b$, $h = f$ and $o = f$	$g = b$, $h = f$ and $o = i$
$g = b$, $h = f$ and $o = j$	$g = b$, $h = f$ and $o = m$	$g = b$, $h = f$ and $o = n$	$g = b$, $h = f$ and $l = f$
$g = b$, $h = f$ and $l = i$	$g = b$, $h = f$ and $l = j$	$g = b$, $h = f$ and $m = j$	$g = b$, $h = f$ and $p = j$
$g = b$, $h = f$ and $p = n$	$g = b$, $k = b$ and $k = f$	$g = b$, $k = b$ and $k = i$	$g = b$, $k = b$ and $k = j$
$g = b$, $h = f$ and $o = f$	$g = b$, $k = b$ and $o = i$	$g = b$, $k = b$ and $o = j$	$g = b$, $k = b$ and $o = m$
$g = b$, $h = f$ and $o = n$	$g = b$, $k = b$ and $l = f$	$g = b$, $k = b$ and $l = i$	$g = b$, $k = b$ and $l = j$
$g = b$, $h = f$ and $m = j$	$g = b$, $k = b$ and $p = j$	$g = b$, $k = b$ and $p = n$	$g = b$, $k = e$ and $k = i$
$g = b$, $k = e$ and $k = j$	$g = b$, $k = e$ and $o = f$	$g = b$, $k = e$ and $o = i$	$g = b$, $k = e$ and $o = j$
$g = b$, $k = e$ and $o = m$	$g = b$, $k = e$ and $o = n$	$g = b$, $k = e$ and $l = f$	$g = b$, $k = e$ and $l = i$
$g = b$, $k = e$ and $l = j$	$g = b$, $k = e$ and $m = j$	$g = b$, $k = e$ and $p = j$	$g = b$, $k = e$ and $p = n$
$g = b$, $k = f$ and $k = j$	$g = b$, $k = f$ and $o = f$	$g = b$, $k = f$ and $o = i$	$g = b$, $k = f$ and $o = j$
$g = b$, $k = f$ and $o = m$	$g = b$, $k = f$ and $o = n$	$g = b$, $k = f$ and $l = f$	$g = b$, $k = f$ and $l = i$
$g = b$, $k = f$ and $o = m$	$g = b$, $k = f$ and $o = n$	$g = b$, $k = f$ and $l = f$	$g = b$, $k = f$ and $l = i$
$g = b$, $k = f$ and $l = j$	$g = b$, $k = f$ and $m = j$	$g = b$, $k = f$ and $p = j$	$g = b$, $k = f$ and $p = n$
$g = b$, $k = i$ and $o = f$	$g = b$, $k = i$ and $o = i$	$g = b$, $k = i$ and $o = j$	$g = b$, $k = i$ and $o = m$
$g = b$, $k = i$ and $o = n$	$g = b$, $k = i$ and $l = f$	$g = b$, $k = i$ and $l = j$	$g = b$, $k = i$ and $m = j$
$g = b$, $k = i$ and $p = j$	$g = b$, $k = i$ and $p = n$	$g = b$, $k = j$ and $o = f$	$g = b$, $k = j$ and $o = i$
$g = b$, $k = j$ and $o = j$	$g = b$, $k = j$ and $o = m$	$g = b$, $k = j$ and $o = n$	$g = b$, $k = j$ and $l = f$
$g = b$, $k = j$ and $l = i$	$g = b$, $k = j$ and $p = n$	$g = b$, $o = f$ and $o = j$	$g = b$, $o = f$ and $o = n$
$g = b$, $o = f$ and $l = i$	$g = b$, $o = f$ and $l = j$	$g = b$, $o = f$ and $m = j$	$g = b$, $o = f$ and $p = j$
$g = b$, $o = i$ and $p = n$	$g = b$, $o = i$ and $o = n$	$g = b$, $o = i$ and $l = f$	$g = b$, $o = i$ and $l = j$
$g = b$, $o = i$ and $m = j$	$g = b$, $o = i$ and $p = j$	$g = b$, $o = i$ and $p = n$	$g = b$, $o = j$ and $o = m$
$g = b$, $o = j$ and $l = f$	$g = b$, $o = j$ and $l = i$	$g = b$, $o = j$ and $m = j$	$g = b$, $o = j$ and $p = n$
$g = b$, $o = m$ and $l = f$	$g = b$, $o = m$ and $l = i$	$g = b$, $o = m$ and $l = j$	$g = b$, $o = m$ and $m = j$
$g = b$, $o = m$ and $p = j$	$g = b$, $o = m$ and $p = n$	$g = b$, $o = n$ and $l = f$	$g = b$, $o = n$ and $l = i$
$g = b$, $o = n$ and $l = j$	$g = b$, $o = n$ and $m = j$	$g = b$, $o = n$ and $p = j$	$g = b$, $o = n$ and $p = n$
$g = b$, $l = f$ and $l = j$	$g = b$, $l = f$ and $m = j$	$g = b$, $l = f$ and $p = j$	$g = b$, $l = f$ and $p = n$
$g = b$, $l = i$ and $m = j$	$g = b$, $l = i$ and $p = j$	$g = b$, $l = i$ and $p = n$	$g = b$, $l = j$ and $p = j$
$g = b$, $l = j$ and $p = n$	$g = b$, $m = j$ and $p = n$	$g = d$, $h = b$ and $h = f$	$g = d$, $h = b$ and $k = e$
$g = d$, $h = b$ and $k = f$	$g = d$, $h = b$ and $k = i$	$g = d$, $h = b$ and $k = j$	$g = d$, $h = b$ and $o = f$
$g = d$, $h = b$ and $o = i$	$g = d$, $h = b$ and $o = j$	$g = d$, $h = b$ and $o = m$	$g = d$, $h = b$ and $o = n$
$g = d$, $h = b$ and $l = f$	$g = d$, $h = b$ and $l = i$	$g = d$, $h = b$ and $l = j$	$g = d$, $h = b$ and $m = j$
$g = d$, $h = b$ and $p = j$	$g = d$, $h = b$ and $p = n$	$g = d$, $h = e$ and $k = b$	$g = d$, $h = e$ and $k = f$
$g = d$, $h = e$ and $k = i$	$g = d$, $h = e$ and $k = j$	$g = d$, $h = e$ and $k = b$	$g = d$, $h = e$ and $o = f$
$g = d$, $h = e$ and $o = i$	$g = d$, $h = e$ and $o = j$	$g = d$, $h = e$ and $o = m$	$g = d$, $h = e$ and $o = n$
$g = d$, $h = e$ and $l = f$	$g = d$, $h = e$ and $l = i$	$g = d$, $h = e$ and $l = j$	$g = d$, $h = e$ and $m = j$
$g = d$, $h = e$ and $p = j$	$g = d$, $h = e$ and $p = n$		

Appendix B

Using the Diagonal Pairs Method for a Constrained 2×2 Puzzle Grid

Figure B.1: A disjoint sub-grid of Fig. 3.10

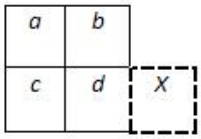

Fig. B.1 shows a disjoint sub-grid of Fig. 3.10 that is constrained by the value placed in cell X. For ease, cells are labeled with and referred to by a unique letter label. It can be noted, by using the notation of Section 3.2.3, that the only negative diagonal pairs is bc. Cells c and d are constrained; they cannot accept the (constant) value placed in X. Therefore, cells a and b are also limited in some cases; they may or may not accept the value in cell X. For example, when $b = c$, cell c cannot accept the fixed value in cell X which implies that cell b also cannot accept the value in cell X. The cases are:

- $b = c$, where $a = X$ or $a \neq X$,
- $b \neq c$ where $a = X$, $b = X$ or $a, b \neq X$.

Working left-to-right and top-to bottom, it can be determined, for each case, how many of the x values can be added to each cell in the puzzle in turn.

- Case 1: $(x-1)(x-2)+(x-1)(x-2)^2$,
- Case 2: $(x-1)(x-2)(x-3)+(x-1)(x-2)^2+(x-1)(x-2)(x-3)^2$.

Hence, following simplification, there are $(x-1)(x-2)(x^2-3x+3)$ valid ways of placing values (from the range $1,\ldots,9$) without duplication.

Appendix C

Using the Diagonal Pairs Method for a 3×3 Puzzle Grid

Figure C.1: A disjoint sub-grid of Fig. 3.9

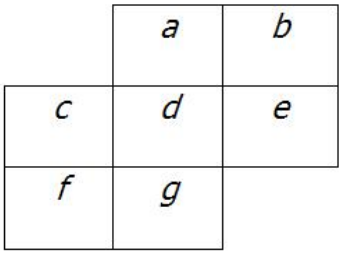

Fig C.1 shows a disjoint sub-grid of Fig. 3.9. For ease, cells are labeled with and referred to by a unique letter label. It can be noted, by using the notation of Section 3.2.3, that the negative diagonal pairs are ca, cb, db, fa, fb, fd, fe, ge and gb. The positive diagonal pairs are ae and cg. Therefore, the cases are:

- $c = a$ only,
- $c = b$ only,
- $d = b$ only,
- $f = a$ only,
- $f = b$ only,
- $f = d$ only,
- $f = e$ only,

- $g = e$ only,
- $g = b$ only,
- $c = a$ and $d = b$ only,
- $c = a$ and $f = b$ only,
- $c = a$ and $f = d$ only,
- $c = a$ and $f = e$ only,
- $c = a$ and $g = e$ only,
- $c = a$ and $g = b$ only,
- $c = b$ and $f = a$ only,
- $c = b$ and $f = d$ only,
- $c = b$ and $f = e$ only,
- $c = b$ and $g = e$ only,
- $c = b$ and $g = b$ only,
- $d = b$ and $f = a$ only,
- $d = b$ and $f = b$ only,
- $d = b$ and $f = d$ only,
- $d = b$ and $f = e$ only,
- $d = b$ and $g = e$ only,
- $f = a$ and $f = e$ only,
- $f = a$ and $g = e$ only,
- $f = a$ and $g = b$ only,
- $f = b$ and $f = d$ only,
- $f = b$ and $g = e$ only,
- $f = d$ and $g = e$ only,
- $f = d$ and $g = b$ only,
- $f = e$ and $g = b$ only,
- $c = a$ and $d = b$ and $f = b$ only,
- $c = a$ and $d = b$ and $f = d$ only,
- $c = a$ and $d = b$ and $f = e$ only,
- $c = a$ and $d = b$ and $g = e$ only,
- $c = a$ and $f = b$ and $f = d$ only,

- $c = a$ and $f = b$ and $g = e$ only,
- $c = a$ and $f = d$ and $g = e$ only,
- $c = a$ and $f = d$ and $g = b$ only,
- $c = a$ and $f = e$ and $g = b$ only,
- $c = b$ and $f = a$ and $f = e$ only,
- $c = b$ and $f = a$ and $g = e$ only,
- $c = b$ and $f = a$ and $g = b$ only,
- $c = b$ and $f = d$ and $g = e$ only,
- $c = b$ and $f = d$ and $g = b$ only,
- $c = b$ and $f = e$ and $g = b$ only,
- $d = b$ and $f = a$ and $f = e$ only,
- $d = b$ and $f = a$ and $g = e$ only,
- $d = b$ and $f = b$ and $f = d$ only,
- $f = a$ and $f = e$ and $g = b$ only,
- $c = a$ and $d = b$ and $f = b$ and $f = d$ only,
- $c = b$ and $f = a$ and $f = e$ and $g = b$ only,
- $d = b$ and $f = b$ and $f = d$ and $g = e$ only,
- $d = b$ and $f = b$ and $f = d$ and $c = a$ and $g = e$ only,
- none of the negative diagonal pairs are equal.

Each case has four sub-cases, based on the positive diagonal pairs.

- $a = e$,
- $c = g$,
- $a = e$ and $c = g$,
- none of the downward-right diagonal pairs are equal.

Working left-to-right and top-to bottom, it can be determined, for each case, how many of the x values can be added to each cell in the puzzle in turn.

- Case 1: $x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- Case 2: $x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)(x-4)$,
- Case 3: $x(x-1)(x-2)(x-3)(x-4)(x-5) + 2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 4: $x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)(x-4)$,
- Case 5: $x(x-1)(x-2)(x-3)(x-4)(x-5) + 2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,

- Case 6: $x(x-1)(x-2)(x-3)(x-4)(x-5) + 2x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 7: $x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)(x-4)$
- Case 8: $x(x-1)(x-2)(x-3)(x-4)(x-5)$,
- Case 9: $x(x-1)(x-2)(x-3)(x-4)(x-5) + x(x-1)(x-2)(x-3)(x-4)$,
- Case 10: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 11: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 12: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 13: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 14: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 15: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 16: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 17: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 18: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 19: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 20: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 21: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 22: 0
- Case 23: 0
- Case 24: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 25: $x(x-1)(x-2)(x-3)(x-4)$
- Case 26: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 27: $x(x-1)(x-2)(x-3)(x-4)$
- Case 28: $x(x-1)(x-2)(x-3)(x-4)$
- Case 29: 0
- Case 30: $x(x-1)(x-2)(x-3)(x-4)$
- Case 31: $x(x-1)(x-2)(x-3)(x-4)$
- Case 32: $x(x-1)(x-2)(x-3)(x-4) + x(x-1)(x-2)(x-3)$,
- Case 33: $x(x-1)(x-2)(x-3)(x-4)$,
- Case 34: 0
- Case 35: 0
- Case 36: $x(x-1)(x-2)(x-3)$,

- Case 37: $x(x-1)(x-2)(x-3)$,
- Case 38: 0
- Case 39: 0
- Case 40: $x(x-1)(x-2)(x-3)$,
- Case 41: $x(x-1)(x-2)(x-3)$,
- Case 42: $x(x-1)(x-2)(x-3)$,
- Case 43: $x(x-1)(x-2)(x-3)$,
- Case 44: $x(x-1)(x-2)(x-3)$,
- Case 45: $x(x-1)(x-2)(x-3)$,
- Case 46: $x(x-1)(x-2)(x-3)$,
- Case 47: $x(x-1)(x-2)(x-3)$,
- Case 48: $x(x-1)(x-2)(x-3)+x(x-1)(x-2)$,
- Case 49: $x(x-1)(x-2)(x-3)$,
- Case 50: $x(x-1)(x-2)(x-3)+x(x-1)(x-2)$,
- Case 51: $x(x-1)(x-2)(x-3)$,
- Case 52: $x(x-1)(x-2)(x-3)(x-4)+2x(x-1)(x-2)(x-3)+x(x-1)(x-2)$
- Case 53: $x(x-1)(x-2)(x-3)$,
- Case 54: $x(x-1)(x-2)(x-3)$,
- Case 55: $x(x-1)(x-2)$,
- Case 56: $x(x-1)(x-2)(x-3)$,
- Case 57: $x(x-1)(x-2)$,
- Case 58: $x(x-1)(x-2)(x-3)(x-4)[(x-5)(x-6)+2(x-5)+1]$.

Hence, combining cases and simplifying, there are $x(x-1)(x-2)(x^4-7x^3+20x^2-28x+17)$ valid ways of placing values (from the range $1,\ldots,9$) without duplication.

Appendix D

Using the Diagonal Pairs Method for a Constrained 3×3 Puzzle Grid

Figure D.1: A disjoint sub-grid of Fig. 3.9

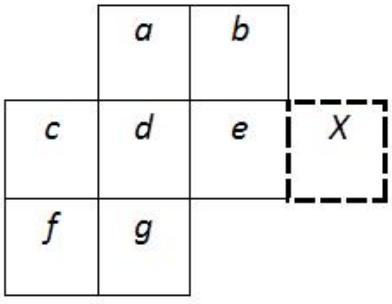

Fig. D.1 shows a disjoint sub-grid of Fig. 3.9, that is constrained by the value placed in cell X. For ease, cells are labeled with and referred to by a unique letter label. It can be noted, by using the notation of Section 3.2.3, that the negative diagonal pairs are ca, cb, db, fa, fb, fd, fe, ge and gb. The positive diagonal pairs are ae and cg. Cells c, d and e are constrained; they cannot accept the (constant) value placed in X. Therefore, cells a and b are also limited in some cases; they may or may not accept the value in cell X. For example, when $b = c$, cell c cannot accept the fixed value in cell X which implies that cell b also cannot accept the value in cell X. The cases are:

- $c = a$ only,
- $c = b$ only,

- $d = b$ only,
- $f = a$ only,
- $f = b$ only,
- $f = d$ only,
- $f = e$ only,
- $g = e$ only,
- $g = b$ only,
- $c = a$ and $d = b$ only,
- $c = a$ and $f = b$ only,
- $c = a$ and $f = d$ only,
- $c = a$ and $f = e$ only,
- $c = a$ and $g = e$ only,
- $c = a$ and $g = b$ only,
- $c = b$ and $f = a$ only,
- $c = b$ and $f = d$ only,
- $c = b$ and $f = e$ only,
- $c = b$ and $g = e$ only,
- $c = b$ and $g = b$ only,
- $d = b$ and $f = a$ only,
- $d = b$ and $f = b$ only,
- $d = b$ and $f = d$ only,
- $d = b$ and $f = e$ only,
- $d = b$ and $g = e$ only,
- $f = a$ and $f = e$ only,
- $f = a$ and $g = e$ only,
- $f = a$ and $g = b$ only,
- $f = b$ and $f = d$ only,
- $f = b$ and $g = e$ only,
- $f = d$ and $g = e$ only,
- $f = d$ and $g = b$ only,
- $f = e$ and $g = b$ only,

- $c = a$ and $d = b$ and $f = b$ only,
- $c = a$ and $d = b$ and $f = d$ only,
- $c = a$ and $d = b$ and $f = e$ only,
- $c = a$ and $d = b$ and $g = e$ only,
- $c = a$ and $f = b$ and $f = d$ only,
- $c = a$ and $f = b$ and $g = e$ only,
- $c = a$ and $f = d$ and $g = e$ only,
- $c = a$ and $f = d$ and $g = b$ only,
- $c = a$ and $f = e$ and $g = b$ only,
- $c = b$ and $f = a$ and $f = e$ only,
- $c = b$ and $f = a$ and $g = e$ only,
- $c = b$ and $f = a$ and $g = b$ only,
- $c = b$ and $f = d$ and $g = e$ only,
- $c = b$ and $f = d$ and $g = b$ only,
- $c = b$ and $f = e$ and $g = b$ only,
- $d = b$ and $f = a$ and $f = e$ only,
- $d = b$ and $f = a$ and $g = e$ only,
- $d = b$ and $f = b$ and $f = d$ only,
- $f = a$ and $f = e$ and $g = b$ only,
- $c = a$ and $d = b$ and $f = b$ and $f = d$ only,
- $c = b$ and $f = a$ and $f = e$ and $g = b$ only,
- $d = b$ and $f = b$ and $f = d$ and $g = e$ only,
- $d = b$ and $f = b$ and $f = d$ and $c = a$ and $g = e$ only,
- none of the negative diagonal pairs are equal.

Each case has four sub-cases, based on the positive diagonal pairs.

- $a = e$,
- $c = g$,
- $a = e$ and $c = g$,
- none of the downward-right diagonal pairs are equal.

Working left-to-right and top-to bottom, it can be determined, for each case, how many of the x values can be added to each cell in the puzzle in turn.

- Case 1: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)$,
- Case 2: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 3: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+2x(x-1)(x-2)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 4: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 5: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+2x(x-1)(x-2)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 6: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+2x(x-1)(x-2)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 7: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+(x-1)(x-2)(x-3)(x-3)(x-4)$
- Case 8: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)$,
- Case 9: $(x-1)(x-2)(x-3)(x-3)(x-4)(x-5)+(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 10: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 11: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 12: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 13: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 14: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 15: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 16: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 17: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 18: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 19: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 20: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 21: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 22: 0
- Case 23: 0
- Case 24: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 25: $(x-1)(x-2)(x-3)(x-3)(x-4)$
- Case 26: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 27: $(x-1)(x-2)(x-3)(x-3)(x-4)$
- Case 28: $(x-1)(x-2)(x-3)(x-3)(x-4)$
- Case 29: 0
- Case 30: $(x-1)(x-2)(x-3)(x-3)(x-4)$
- Case 31: $(x-1)(x-2)(x-3)(x-3)(x-4)$

- Case 32: $(x-1)(x-2)(x-3)(x-3)(x-4)+(x-1)(x-2)(x-3)(x-3)$,
- Case 33: $(x-1)(x-2)(x-3)(x-3)(x-4)$,
- Case 34: 0
- Case 35: 0
- Case 36: $(x-1)(x-2)(x-3)(x-3)$,
- Case 37: $(x-1)(x-2)(x-3)(x-3)$,
- Case 38: 0
- Case 39: 0
- Case 40: $(x-1)(x-2)(x-3)(x-3)$,
- Case 41: $(x-1)(x-2)(x-3)(x-3)$,
- Case 42: $(x-1)(x-2)(x-3)(x-3)$,
- Case 43: $(x-1)(x-2)(x-3)(x-3)$,
- Case 44: $(x-1)(x-2)(x-3)(x-3)$,
- Case 45: $(x-1)(x-2)(x-3)(x-3)$,
- Case 46: $(x-1)(x-2)(x-3)(x-3)$,
- Case 47: $(x-1)(x-2)(x-3)(x-3)$,
- Case 48: $(x-1)(x-2)(x-3)(x-3)+(x-1)(x-2)(x-3)$,
- Case 49: $(x-1)(x-2)(x-3)(x-3)$,
- Case 50: $(x-1)(x-2)(x-3)(x-3)+(x-1)(x-2)(x-3)$,
- Case 51: $(x-1)(x-2)(x-3)(x-3)$,
- Case 52: $(x-1)(x-2)(x-3)(x-3)(x-4)+2x(x-1)(x-2)(x-3)+(x-1)(x-2)(x-3))$,
- Case 53: $(x-1)(x-2)(x-3)(x-3)$,
- Case 54: $(x-1)(x-2)(x-3)(x-3)$,
- Case 55: $(x-1)(x-2)(x-3)$,
- Case 56: $(x-1)(x-2)(x-3)(x-3)$,
- Case 57: $(x-1)(x-2)(x-3)$,
- Case 58: $(x-1)(x-2)(x-3)(x-3)(x-4)[(x-5)(x-6)+2(x-5)+1]$.

Hence, combining the cases and simplifying, there are $(x-1)(x-2)(x-3)(x^4-7x^3+20x^2-28x+17)$ valid ways of placing values (from the range $1,\ldots,9$) without duplication.